



Joe St Sauver, Ph.D.

Distinguished Scientist and Director of Research, Farsight Security, Inc.

“Bang_Question”

A Tutorial Proof-of-Concept Cyber Investigative Framework



"Bang_Question:" A Tutorial Proof-of-Concept Cyber Investigative Framework

Version 1.1
March 2021

Joe St Sauver, Ph.D.
<stsauver@fsi.io>
Distinguished Scientist and Director of Research
Farsight Security, Inc.

Copyright(c) 2021, Farsight Security, Inc.

TLP **White** : Disclosure is not limited.

*On the "bangquestion" name and icon: Some may wonder where the program's title and icon came from. The answer is simple: often systems and networks will be running normally when suddenly something "flies in from left field" and "bangs into things (!)". That may leave you wanting to know "Now what the heck was **THAT?**" ("?") The application name & icon seemed to capture that "bang_question" experience perfectly.*

TABLE OF CONTENTS

PART I. TEST APP WITH THREE STATIC TABS

1. INTRODUCTION	5
2. THE INVESTIGATIVE PROCESS: GOING FROM ARTISANAL CRAFTSMANSHIP TO THE ASSEMBLY LINE	6
3. A QUICK NOTE ABOUT PYTHON VERSIONS AND TKINTER	8
4. GETTING THE OVERARCHING TKINTER NOTEBOOK STRUCTURE CREATED	8
5. TWEAKING THE STYLE A LITTLE	10
6. ADDING A MENUBAR, HANDLING CALLBACKS AND DOING A LITTLE INITIAL TESTING	12
7. ADDING A LOG AREA FOR INFORMATIONAL NOTES	14
8. SNAGGING A COPY OF A WEBPAGE (IN GENERAL)	15
9. USING PYPETEER TO CAPTURE SCREEN SHOTS	18
10. DISPLAYING A SCROLLABLE JPEG IN TKINTER	22
11. FILLING IN PASSIVE DNS RRNAME DATA ON OUR THIRD SAMPLE TAB	23
12. A QUICK CHECKPOINT BEFORE WE GO ON	24

PART II. FULL PROOF-OF-CONCEPT APP WITH DYNAMIC TABS

13. TAB MANAGEMENT AND HANDLING MULTIPLE RUNS	34
14. SCRAPING A PAGE	48
15. DOING OUR DNSDB QUERIES	50
16. HANDLING THE CONTENT FOR THE ASN WHOIS TAB	55
17. HANDLING DRAWING THE NETWORK GRAPH (THE CONTENT FOR THE FINAL TAB)	58
18. REMAINING ISSUES	60

APPENDICIES

I. INSTALLATION INSTRUCTIONS	66
II. FULL SOURCE CODE FOR THE FULL PROOF-OF-CONCEPT APPLICATION	68
III. STANDARD LIBRARIES AND THIRD-PARTY PACKAGES	87
IV. LICENSES AND ACKNOWLEDGEMENTS	89
V. SCHOOL WEB PAGES REVIEWED FOR HEIGHT MEASUREMENT PURPOSES	90

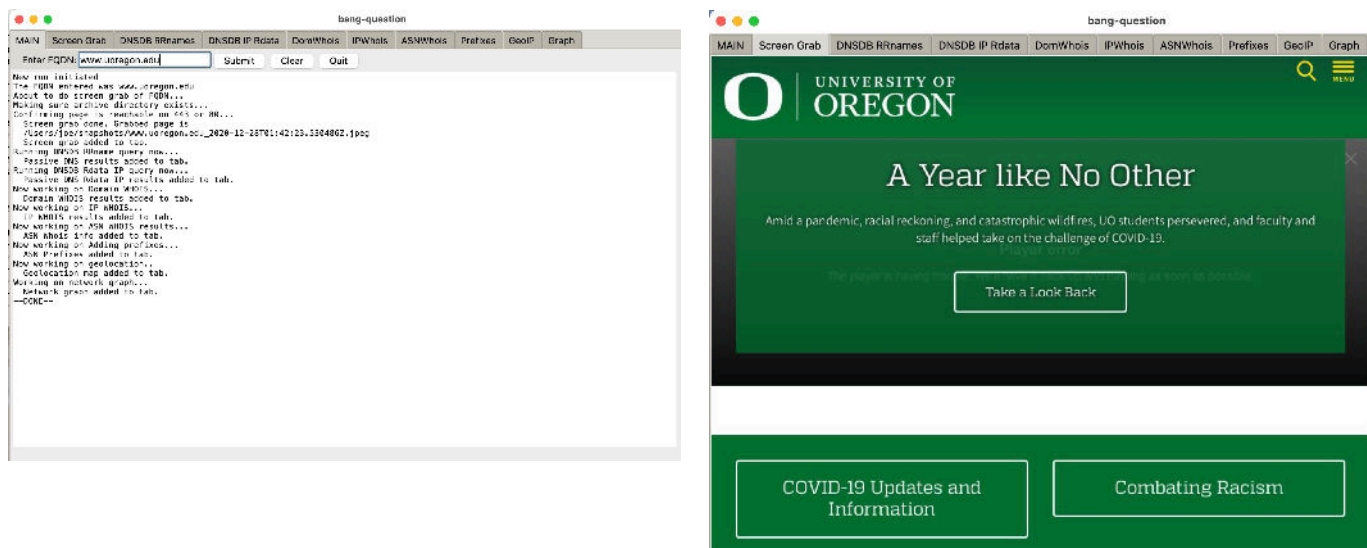
PART I.

TEST APP WITH THREE STATIC TABS

1. INTRODUCTION

WHO? The audience for this report consists of subject matter experts and cyber security analysts interested in learning how to script analyses of fully qualified domain names (FQDNs).

WHAT? In the body of this report, we'll dig into the details, but for now, the "10,000 foot" overview is that when you run the proof-of-concept application, a graphical investigative "notebook" will launch. After you submit a FQDN that you're interested in, the application will do a series of automated checks, including attempting to capture a picture of what that site looks like, checking passive DNS for the domain name (assuming you've got a DNSDB API key), checking Domain/IP/ASN WHOIS, geolocating the IP, etc. For example, here's what the app's opening (main) tab and the app's (scrollable) screen grab tab look like for the university site at www.uoregon.edu (sample output available from the other eight tabs omitted here):



WHY? We created this application for two reasons: first, the application can conveniently and efficiently perform the same routine checks that a trained investigator would normally do manually -- we're just eliminating the tedious grunt work associated with that process. Second, beyond providing a pragmatically-useful tool as-shipped, we also wanted to demystify the process and show you HOW we built this tool, so you can extend it with additional checks you may routinely employ (but which we haven't implemented).

HOW? We built this open-source framework using Python3, Tkinter and a variety of open source libraries. Details of how we did it and what we learned while doing it can be found in the body of this report.

NOTE: If you don't care about any of the "blah blah blah", and just want installation instructions and the code for the application, see Appendix I and Appendix II of this report or, for the latest code, see https://github.com/farsightsec/blog-code/tree/master/bang_question

NOTICE: This is an experimental/proof-of-concept/under-development application. It has known and unknown flaws and limitations and comes with **ABSOLUTELY NO WARRANTY**. If you choose to use it, you do so at your own discretion and agree to assume any/all risks, if any, associated with doing so. You also accept sole responsibility for any sites you access, and any copyrighted or proprietary data you may retrieve from them.

2. THE INVESTIGATIVE PROCESS: GOING FROM ARTISANAL CRAFTSMANSHIP TO THE ASSEMBLY LINE

Most subject matter experts have a fairly-well-standardized approach to reviewing domains of interest:

- If a site is unknown (but not believed to be particularly attuned to unexpected visitors) a common first step might be to "eyeball" the site from a sandboxed browser. The analyst may be looking to see:
 - Is the site perhaps impersonating a well-known bank or payment card company?
 - Is it attempting to sell something illegal, such as scheduled controlled substances or knock-off brand name merchandise?
 - Or is the site just a regular website, perhaps used by some small business, a school, a church, or some other organization?
- Next, an analyst will often review the site in Farsight Security's DNSDB Passive DNS (subscription required):
 - Where has the site been hosted over time? Was it on just one IP address for a long time? Or has the site been continually hopping from IP to another?
 - What other sites (if any) share the site's current IP address? If that search radius is expanded, is the neighborhood "typical," or are there a lot of other sketchy-sounding hosts also hosted nearby?
 - Does the site use a shared name server or mail server? Does that shared resource lead to any other interesting discoveries, either in terms of related bad sites or sites that may help with identifying the owner of the primary site of interest?
- Most analysts will also routinely check WHOIS:
 - What does WHOIS for the domain name look like? Is it one that's existed for a long time, or was it just registered? Does it use a highly regarded registrar, or one particularly popular with miscreants? Is the domain registered to a well-known company? Or is the domain owner hiding behind a proxy/privacy registration service?
 - How about WHOIS for the IP address that the domain's currently on? (Even if Domain WHOIS has largely been gutted, IP WHOIS remains surprisingly useable except for some reverse proxy services and cloud providers)
 - How about WHOIS for the AS (Autonomous System) that's announcing/routing that IP? Are there other network prefixes announced by that same ASN? Or does the ASN have just a single new small block?
- Where is the site's current IP address located, geographically? Is it in the United States? Overseas?
- What does a graphical network diagram of the DNSDB RRname results look like?
- Is the current site listed on one or more blocklists? Conversely, is it listed on any allowlists? Are there spam, phishing, or malware reports related to the domain or the IP it's using?
- Can we find linkages to other relevant domains by looking at the names on the site's SSL/TLS certificate?

The exact set and sequence of steps an analyst follows, and how they interpret what they see, will vary from analyst to analyst. The point is not the particular steps that a given analyst employs, or even how they interpret what they see, but the fact that these are steps that are *routinely performed*. Any time you hear that something is being "routinely performed," that's the signal that there may be potential for automation. Rather than manually performing a routine set of steps time after time after time, why not let a computer do that

"grunt work" for you automatically and consistently? Often the answer may be, "Well, we'd love to do that -- we just don't know how to automate our processing!"

This document is meant to help teach you to automate your domain analyses. We're going to show you an example of how to construct a GUI proof-of-concept analysis framework using Python with Tkinter.

Note that our `bang_question` application isn't a "polished" and "bullet proof" commercial product, but more of a quick-and-dirty demonstration of the "art of the possible." It doesn't handle "everything" (not even the full set of analyses mentioned in the bulleted list above!), but it does show examples of handling diverse data sources, and we think you'll find it easy enough to extend and improve the application to meet your needs.

If you aren't interested in investing that "sweat equity," we totally understand. Obviously, you should feel free to continue using established manual processes, or to purchase a polished and commercially supported alternative -- whatever works best for you works great for us too.

For the purpose of this proof of concept, we're just going to take a single interactively-entered fully qualified domain name (FQDN), and then let our program produce a set of nine representative "notebook" pages:

1. A scrollable screen grab of the Fully Qualified Domain Name (FQDN) [displaying JPEG data]
2. A DNSDB RRname Search of the FQDN [reformatting and displaying JSON text-format data]
3. A DNSDB Rdata Search for FQDN --> IP [ditto]
4. A check of the Domain WHOIS for the FQDN [displaying raw JSON data]
5. A check of the IP WHOIS for the IP the FQDN is currently using [ditto]
6. A check of the ASN WHOIS for ASN announcing the IP that the FQDN is currently using [converting XML to HTML and displaying the HTML as a JPEG screen shot]
7. A list of the prefixes associated with that ASN [displaying raw text data]
8. A map showing the location associated with the IP address the FQDN is currently using [displaying a JPEG]
9. A network graph showing the DNSDB RRname search results [displaying a JPEG]

Obviously, that's not all the different things an analyst might do, but it's enough to give you a sense of what's possible.

If we had to build code to do all those tasks from scratch, that would be quite a project. Fortunately, we can leverage a variety of Python3 third-party libraries (see the list of libraries in Appendix III).

What we're going to do for the remainder of this report is introduce you to Tkinter and explain how we build up the overall notebook framework, the main tab, and then each of the other nine tabs. We'll also briefly discuss what an analyst might be substantively looking for in each tab's content.

This document is also meant to capture the oddities we ran across along the way -- quirks of Tkinter and other libraries, plus data oddities, and even policy issues we ran into and resolved (or failed to resolve).

3. A QUICK NOTE ABOUT PYTHON VERSIONS AND TKINTER

This project is explicitly **Python3-only**, since Python2 went end of life January 1st, 2020.¹ In fact, we'll go beyond that and say that this project is only meant to run on the **current version of Python 3_** which means Python 3.9.1 as of the time this was written. Why 3.9.1 or later? Well, as stated in the release notes,² "3.9.1 is the first version of Python to support macOS 11 Big Sur." That's what we're developing on these days.

We also want to stress that we're using Python 3.9.1 **as downloaded from the main Python website**.³ This is important because some Python installations intentionally omit Tkinter, the native Python GUI packages. We need to have Tkinter available for this project. To ensure we get the "right" Python, we'll be explicitly referring to `/usr/local/bin/python3` in our code since `/usr/local/bin/python3` is where we have Python 3.9.1 installed on our system. Your path may vary. You can check your version of Python at that location with:

```
$ /usr/local/bin/python3 --version
Python 3.9.1
```

4. GETTING AN OVERARCHING TKINTER NOTEBOOK STRUCTURE CREATED

We begin by considering the application's Tkinter notebook structure. At its' most basic, we can create a basic Tkinter notebook layout with just a small chunk of code.

In our real application, we'll have a total of ten tabs, and we'll "create them as we need them" rather than building them up from, but you can get a sense of how this all works with just a small test program using three tabs:

```
$ cat test-notebook.py
#!/usr/local/bin/python3

from tkinter import *
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
mywindow = tk.Frame(root)
mynotebook = ttk.Notebook(mywindow)

# colors: www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter
t1 = tk.Frame(height=200,width=200,background='light goldenrod')
t2 = tk.Frame(height=200, width=200,background='PaleTurquoise1')
t3 = tk.Frame(height=200, width=200,background='plum2')

mynotebook.add(t1, text='Tab #1')
mynotebook.add(t2, text='Tab #2')
mynotebook.add(t3, text='Tab #3')
```

¹ <https://www.python.org/doc/sunset-python-2/>

² <https://www.python.org/downloads/release/python-391/>

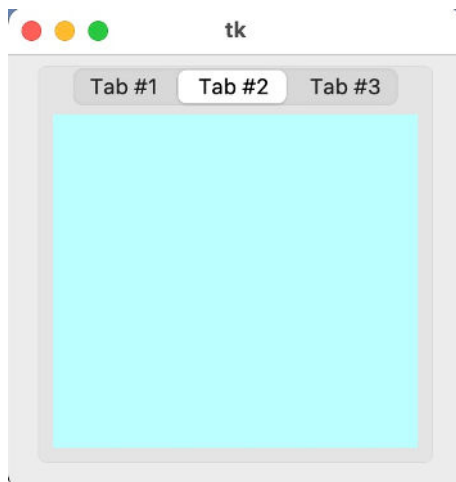
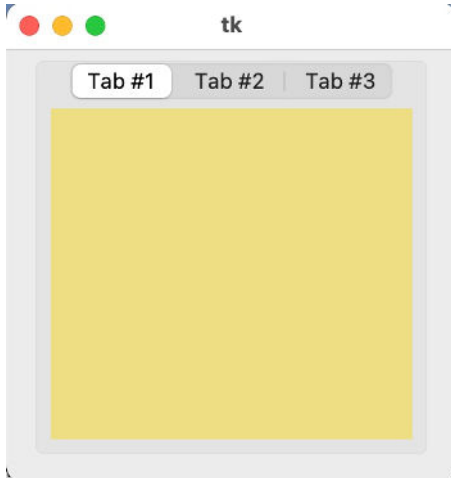
³ <https://www.python.org/downloads/>

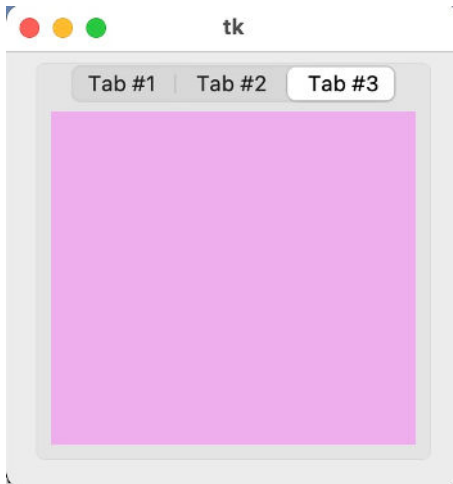

```
mynotebook.pack(expand=True, side=TOP, anchor=NW)
mywindow.pack(side=TOP, anchor=NW)
mywindow.mainloop()
```

We're now ready to make that code executable and actually try running it. We'll do that by saying:

```
$ chmod a+rx test-notebook.py
$ ./test-notebook.py
```

This will cause a small "notebook" to be created and run, with each of the frames in the tabs simply being a different color. We can try clicking through the three tabs to see this:





A few quick notes about that sample code:

- The import statements are used to bring in the libraries we'll be using. As part of that, we can set up shortened references (for example, the shorthand "tk" can be used instead of "tkinter" and "ttk" can be used instead of "tkinter.ttk")
- The widgets we're working with (frames and a notebook) are part of a tree structure, so each will always have a "parent," culminating ultimately in the apex root window.
- The "pack" statements (near the end of that code) handle automatically arranging the elements, eliminating the need for either "hard-coding" widget locations using screen coordinates or the potential use of a grid reference system for relative widget placement.
- Normally we won't use colors as part of our design (perhaps the influence of our earlier days on a monochrome NeXT cube⁴), but in this case setting the widget's color represents an easy way to quickly see that we really are looking at three different tabs.

5. TWEAKING THE STYLE A LITTLE

We also want to change the look and feel of our app a little:

- We want to set the root window title -- its default value is just "tk" (as shown in the screenshots above)
- We also want to rereplace the default icon (the default is a stylized Python "rocket ship"):



- Let's also change the overall "ttk.Style" (we prefer the more rectangular "clam" style).

Admittedly these are only cosmetic tweaks, but they're easily enough made. We'll add the following commands immediately after "root = tk.Tk():"

⁴ <https://en.wikipedia.org/wiki/NeXTcube>

```

root.title("bang_question")
root.tk.call('wm', 'iconphoto', root._w, PhotoImage(file='exclamationquestion.gif'))
s = ttk.Style()
s.configure('TNotebook', tabposition='nw')

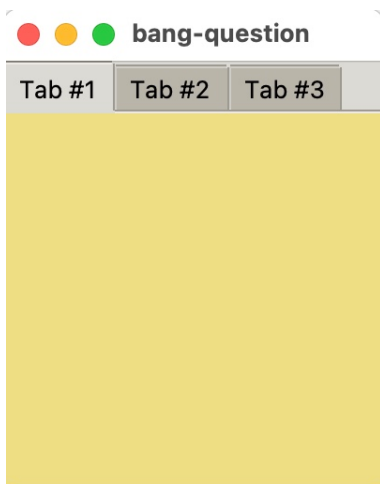
# themes discussed and described: https://tkdocs.com/tutorial/styles.html
# to see a full list of available themes: print(s.theme_names())
s.theme_use('clam')

```

When we rerun the updated application, we can immediately see the new icon:



Our title has been updated, and now our tabs now look "more rectangular", too:



6. ADDING A MENUBAR, HANDLING CALLBACKS AND DOING A LITTLE INITIAL TESTING

You may wonder how you can interact with that application. Currently you can click on one of the tabs (to select that tab), or you can click on one of the operating system menu bar "balls" in the upper left corner of the window (to kill, iconify, or go full screen mode). That's about it.

We need a way to enter a FQDN and then submit it. Let's also explicitly add a way to "clear" (or "reset") things, and a way to quit the application. We'll do that with a small menubar. Let's assume it's going to be part of the first tab, t1, in our notebook:

```

mymenubarbox = ttk.Frame(t1)
mymenubarbox.pack(side=TOP, anchor=NW)

mymenubarlabel = ttk.Label(mymenubarbox, text="    Enter FQDN:")
mymenubarlabel.pack(side=LEFT)

FQDN = ttk.Entry(mymenubarbox)
FQDN.pack(side=LEFT)

buttona = tk.Button(mymenubarbox, text='Submit', command = funca)
buttona.pack(side=LEFT)

```

```

root.bind('<Return>', funca)

buttonb = tk.Button(mymenubarbox, text='Clear', command = funcb)
buttonb.pack(side=LEFT)

buttonc = tk.Button(mymenubarbox, text='Quit', command = funcb)
buttonc.pack(side=LEFT)

mymenubarbox.pack(side=TOP, anchor=NW)

```

By packing "side=LEFT", we'll get a horizontal menubar. We'll insert the new code shown above in our sample file just after the "t3 = tk.Frame" line.

Notice those three buttons refer to three commands (funca, funcb, and funcb). We'll also need to define those callbacks, so the application knows what to do when one of those buttons gets pressed. We'll add those callback functions to the main program, after the import statements, but before the bulk of our main program. For now, we'll keep those callback functions quite simple:

```

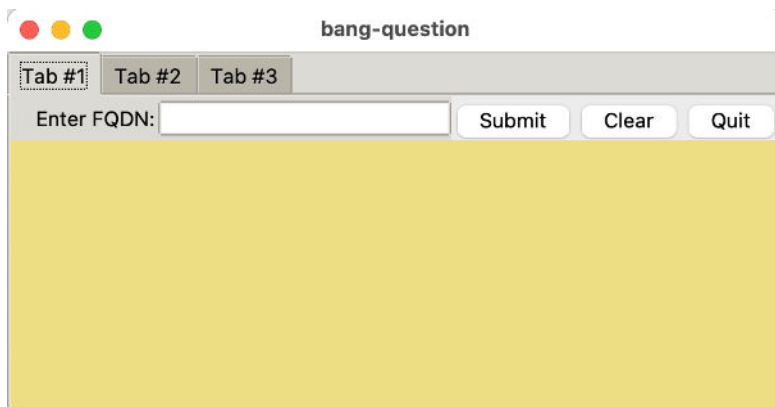
def funca(event=None):
    fqdn = FQDN.get()
    print ("In funca, fqdn="+fqdn)

def funcb(event=None):
    print ("In funcb")
    FQDN.delete(0, 'end')

def funcb(event=None):
    sys.exit()

```

When we run our code now, it looks like:



You'll notice that the window has automatically resized, becoming wider to accommodate the width of the menubar. If we enter a domain name and try pressing "Submit," things don't work quite the way we expected:

```

$ ./test-notebook.py
In funca, fqdn=

```

In this case, funca doesn't seem to be able to "see" fqdn. In retrospect, that should not be surprising since fqdn is a **locally-scoped variable** by default. If we want the contents of that variable to be available to subroutines, we need to either pass that variable as an explicit parameter (and then explicitly return any

changed values), or we need to define those variables as global. Because this code isn't very long or complex, we elect to do the later. We add the statement:

```
global FQDN, fqdn, t2, t3
```

We also add a copy of that global statement in the subroutines that need to be able to see AND MODIFY that variable. Our code now looks like:

```
#!/usr/local/bin/python3

import asyncio
import sys

from tkinter import *
import tkinter as tk
import tkinter.ttk as ttk

global FQDN, fqdn, t2, t3

def funca(event=None):
    global FQDN, fqdn
    fqdn = FQDN.get()
    print ("In funca, fqdn="+fqdn)

def funcb(event=None):
    global FQDN
    print ("In funcb")
    FQDN.delete(0, 'end')

def funcc(event=None):
    sys.exit()

root = tk.Tk()
root.title("bang_question")
root.tk.call('wm', 'iconphoto', root._w, PhotoImage(file='exclamationquestion.gif'))

s = ttk.Style()
s.configure('TNotebook', tabposition='nw')
# to see a full list of potential themes: print(s.theme_names())
s.theme_use('clam')

mywindow = tk.Frame(root)
mynotebook = ttk.Notebook(mywindow)

# colors: www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter
t1 = tk.Frame(height=200,width=200,background='light goldenrod')
t2 = tk.Frame(height=200, width=200,background='PaleTurquoise1')
t3 = tk.Frame(height=200, width=200,background='plum2')

mymenubarbox = ttk.Frame(t1)
mymenubarbox.pack(side=TOP, anchor=NW)

mymenubarlabel = ttk.Label(mymenubarbox, text="    Enter FQDN:")
mymenubarlabel.pack(side=LEFT)

FQDN = ttk.Entry(mymenubarbox)
FQDN.pack(side=LEFT)
```

```

buttona = tk.Button(mymenubarbox, text='Submit', command = funca)
buttona.pack(side=LEFT)
root.bind('<Return>', funca)

buttonb = tk.Button(mymenubarbox, text='Clear', command = funcb)
buttonb.pack(side=LEFT)

buttonc = tk.Button(mymenubarbox, text='Quit', command = funcc)
buttonc.pack(side=LEFT)

mymenubarbox.pack(side=TOP, anchor=NW)

mynotebook.add(t1, text='Tab #1')

mynotebook.add(t2, text='Tab #2')
mynotebook.add(t3, text='Tab #3')

mynotebook.pack(expand=True, side=TOP, anchor=NW)
mywindow.pack(side=TOP, anchor=NW)
mywindow.mainloop()

```

When we try running the code again, entering `www.uoregon.edu` into our new FQDN box, and then hitting Submit, things now work as expected. Moving on to try the Clear button, we see the FQDN box go blank and "In funcb" gets printed out, also as expected. Finally, clicking Quit also works, shutting down the application. So far everything is as expected.

7. ADDING A LOG AREA FOR INFORMATIONAL NOTES

As the program runs, it should keep the user informed about what's happening. We'll handle that by adding a text area "log box" beneath our menubar on tab t1. Doing that only requires a few lines of code. We'll add the required lines immediately below the `"mymenubarbox.pack(side=TOP, anchor=NW)"` line:

```

my_log_box = tk.Text(t1,height=40,width=132)
my_log_box.pack(side=TOP, anchor=NW)

```

We also need to declare that widget as global:

```

global my_log_box

```

When we want to add an entry to that box, we'll use commands like:

```

my_log_box.insert(tk.END, "Here's a note to add to the log box.\n")
my_log_box.update()

```

The reference to `"tk.END"` ensures we write to the end of any existing text in the log box.

The `"my_log_box.update()"` command ensures that the log entry gets displayed immediately (it might otherwise end up buffered and not get displayed for some time).

8. SNAGGING A COPY OF A WEBPAGE (IN GENERAL)

Let's now work on our first "substantive" content, capturing (and then displaying) an image of a web page in our application. Scraping and saving a copy of a webpage can be an important part of an investigation. Why? Well, what you see the *first time* you visit a page may not be what you see when you revisit that same site later. Therefore, it is (or it should be!) a "best practice" to capture what you run into on a web page for potential "evidentiary" purposes.⁵ If you were working manually, you might visit a site with your web browser and then manually take screen shots of what you see. When automating that process, you'll typically use a so-called "**headless browser**." The first question to deal with is, "WHICH headless browser should I use?" Three commonly mentioned options are (1) PhantomJS, (2) Selenium and (3) Pypeteer.

(1) Upon review, we ruled out using **PhantomJS** since the project team suspended further project development in March of 2018.⁶

(2) **Selenium**⁷ is another commonly mentioned option, but in our testing, Selenium had two main drawbacks:

- In Selenium, it can be hard to tell when a page has "finished loading" and is "ready to be captured." This is normally "worked-around" on an "emprical/ad-hoc basis" by adding "delay time." Getting that time right can be tricky, particularly if diverse categories of pages are being captured. The issue is easy to explain:
 - Set it too short? A slow-to-transfer-and-render page may not be "fully settled" when you try capturing a copy of that page as an image.
 - Set it too long? Users may feel as if your application is "slow" (when it fact you're just being careful to let the target web page completely finish loading).
- If you want to get an ENTIRE web page, particularly a long/tall web page that extends beyond what might be visible in a single fixed-sized "viewport", Selenium typically requires stitching together multiple viewport-sized screen captures, viewportful-at-a-time. That can be tedious/inconvenient. You can find a variety of solutions for attempting this if you search the web for

`selenium full page screenshot +python`

(3) Our third option's **Pypeteer**,⁸ the Python implementation of Puppeteer.

⁵ We use the term "evidentiary" here very loosely. In reality, if you're collecting screen captures for use in criminal prosecutions or civil lawsuits, consideration probably needs to be given to things like logging and timestamping the captures, hashing the capture to prove an absence of tampering, documenting the URL visited, documenting the chain of custody around the captures, etc., etc., etc. These are all potentially quite subtle issues that are out-of-scope for a simple proof of concept application like this one. This is NOT a litigation support application!

⁶ <https://phantomjs.org/>

⁷ <https://www.selenium.dev/>

⁸ See <https://github.com/pypeteer/pypeteer> and <https://pypi.org/project/pypeteer/>

Note <https://github.com/pypeteer/pypeteer/projects> mentions migrating to the "pypeteer2 namespace", see <https://pypi.org/project/pypeteer2/>

Given the issues with PhantomJS and Selenium, we decided we'd rely on Pypeteer for our screen capture requirements. We're not the only one who's been impressed by it. For example, one reviewer stated:⁹

"The Google Chrome team made waves last year when it released Puppeteer, a NodeJS API for running headless Chrome instances. It represents a marked improvement both in terms of speed and stability over existing solutions like PhantomJS and Selenium, and was named one of the ten best web scraping tools of 2018."

Our first Pypeteer hands-on "evaluation task" was using Pypeteer to check to see if modern web pages actually **ARE** long. Perhaps modern web designers are carefully-crafting pages to no more than a minimalist 800 x 600 pixel format? and thus there's no need to worry about scraping long/tall web pages?

For example, what do we see if we look at the home pages of colleges and universities? How tall do they tend to be? We checked 96 colleges/universities using Pypeteer. ALL of those sites had long/tall pages when we visited them with Pypeteer set to a target page width of 800 pixels and a default user agent. We did NOT check for sites with "responsive designs," e.g., sites that change their layout depending on whether a visitor is apparently coming from a large desktop workstation, a laptop computer, a tablet or a phone.

See Appendix V for a full list of the sites we checked and the home page dimensions we observed for those sites. Some of the things we learned included:

- While all of the screen captures were meant to be normalized to a common width of 800 pixels, some sites were apparently fixed width (or somehow otherwise ended up being captured at a width wider than 800 pixels.) This was unexpected.
- The days of having a home page that fits on a single screen (and which doesn't routinely require vertical scrolling to use) appear to be over. If you were to just capture a fixed image footprint by default, you might miss 90% of some home pages! (We also can't help but wonder -- do most visitors actually scroll down on those sort of sites? Or does everything "below the fold" simply get overlooked by visitors "clicking through"? Put another way, how "long/tall" is TOO "long/tall"?)
- 18 out of 96 total sites (~19%) failed to capture successfully when we attempted to scrape them with Pypeteer. Breaking that down:
 - 13/18 appear to use a page design that includes technology (perhaps a video or animation?) that interferes with getting a clean capture of the page. We tried disabling animations with

```
await page._client.send('Animation.disable')
```

but that doesn't appear to have any effect -- some animations still seemed to run. This issue appears to be a known/open Puppeteer bug.¹⁰ We might be able to spend some time tweaking Pypeteer to cleanly capture those, but since this is just a proof-of-concept application, we're not going to bother.

⁹ <https://hackernoon.com/tips-and-tricks-for-web-scraping-with-puppeteer-ed391a63d952>

¹⁰ <https://github.com/puppeteer/puppeteer/issues/511>

- 2/18 sites "capture" as a totally blank (white) page for as-yet-undetermined reasons -- maybe an issue with transparency?
- 1/18 sites had a problem with a fundraising-related overlay obfuscating the underlying page. Manual visitors to that site would just click through the overlay to close it, but our automated scraping just ended up capturing a copy of the obscuring overlay instead of the underlying page, very irritating.
- 1/18 totally failed to successfully scrape due to the presence of a redirection loop. We can replicate the looping issue if we attempt to visit the site using the curl command line client with the "-L" ("follow redirects") option active:

```
$ curl -L https://www.osu.edu/
curl: (47) Maximum (50) redirects followed
```

Using curl *without* the -L ("follow redirects") option, we can see that the site appears (for some reason) to be trying to redirect "from itself" "to itself", an obvious recipe for failure:

```
$ curl https://www.osu.edu/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://www.osu.edu/">here</a>.</p>
</body></html>
```

- At least 1/96 sites didn't fail, but only because we'd intentionally disabled SSL/TLS certificate checking. That site, <https://www.oregonstate.edu/>, used and uses a cert that doesn't include www.oregonstate.edu as a certificate Subject Alternative Name. Again, we can use curl at the command line to observe this:

```
$ curl https://www.oregonstate.edu/
curl: (60) SSL: no alternative certificate subject name matches target host name
'www.oregonstate.edu'
More details here: https://curl.haxx.se/docs/sslcerts.html
```

- In general, we might expect that criminal or malicious sites (unlike the regular college and university web sites we tested) won't necessarily "play nice" when it comes to facilitating screen captures. For example, we believe some criminal or malicious sites may intentionally use/attempt to use a variety of techniques to frustrate automated screen captures. This may include intentionally incorporating animation, using overlays, using Captchas, sniffing browser agents, etc. Thus, capturing a copy of a site with Pypeteer will OFTEN work, but SHOULD'N'T be counted on to **ALWAYS** work. ***Please promptly review any screen captures you perform to ensure that they'll meet your needs (and so you can manually recapture them if that's necessary).***

9. USING PYPETEER TO CAPTURE SCREEN SHOTS

In order to be able to use Pypeteer, we'll need to install it. We'll install it (and other subsequent third-party libraries) using pip3:¹¹

```
# /usr/local/bin/pip3 install pypeteer
```

We'll also need to do a one-time install of Chromium. Pypeteer provides a convenience function for this:

```
$ pypeteer-install
```

We're then ready to incorporate Pypeteer into our application.

The actual code we need to scrape the page amounts to only a few lines, a couple of imports:

```
import asyncio
from pypeteer import launch
```

and then:

```
browser = await launch({'headless': True, 'ignoreHTTPSErrors': True, \
    'defaultViewport': None, 'viewport_width': 800})
context = await browser.createIncognitoBrowserContext()
page = await browser.newPage()
await page._client.send('Animation.disable')
await page.goto(url)
await page.screenshot({'path': myfilespec, 'type': 'jpeg', \
    'quality': 80, 'fullPage': True})
await browser.close()
```

Before running that code, there are a few other things we should mention:

- We need to make sure the site we want to go to actually exists and is reachable on either port 443 (https) or port 80 (http). We'll check the https port first, then fall back to plain old standard http if we have to. If neither work, we'll pop up an error message since this is usually a sign of a typo or other operator error.

This is the code that will handle that check and which will potentially trigger that popup:

```
# verify page exists, and figure out if https is supported
try:
    # note: you do NOT need to explicitly permit inbound traffic on macOS!
    myaddrinfo = socket.getaddrinfo(fqdn, 443)
    url = "https://" + fqdn
    domain_resolves_ok = True
except socket.gaierror:
    # couldn't connect on default SSL/TLS port; fall back to regular http
    try:
        myaddrinfo = socket.getaddrinfo(fqdn, 80)
        url = "http://" + fqdn
        domain_resolves_ok = True
```

¹¹ <https://docs.python.org/3/installing/index.html>

```

except:
    # couldn't even connect on standard port
    display_error()
    return(1)

```

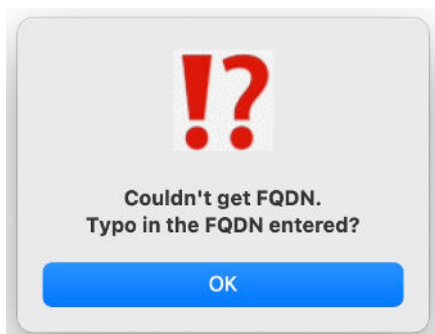
Here's the `display_error` error notification routine:

```

def display_error():
    showerror(message=\
        "Couldn't get FQDN.\nTypo in the FQDN entered?",
        title="Message Box", icon="error")

```

When executed, it will pop up a new window:



As noted in the comments in the code, if you're running this code on macOS, you may at least temporarily see a system prompt asking if you want to allow inbound traffic. You do NOT need to grant this permission for the application to work.

Since we want to snag a copy of the capture, we'll also need to confirm the existence of the directory where we're going to stash those captures. If the directory doesn't exist, we'll need to create it. We also confirm that the screen capture file can be successfully created:

```

import asyncio
import datetime
import errno
import os
from pathlib import Path
import socket
[...]

home = str(Path.home())
subdir = "snapshots"
myfilename = fqdn
utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'
myextension = "jpeg"
mydir = home + "/" + subdir
fullfilenamepart = fqdn + "_" + utcdatetime + "." + myextension
latestversion = fqdn + "_" + "latest" + "." + myextension
[...]

# ensure the directory exists
try:
    os.makedirs(mydir)

```

```

except OSError as e:
    if e.errno != errno.EEXIST:
        raise

# ensure the timestamped file exists
myfilespec = mydir + "/" + fullfilenamepart
try:
    open(myfilespec, 'a').close()
except:
    raise

```

- In addition to checking/creating the capture file directory and filename, we're also going to create a "_latest" "convenience link" for each site to simplify referring to the most recent capture for that site:

```

# ensure the "latest" version is updated for this URL
mylatest = mydir + "/" + latestversion

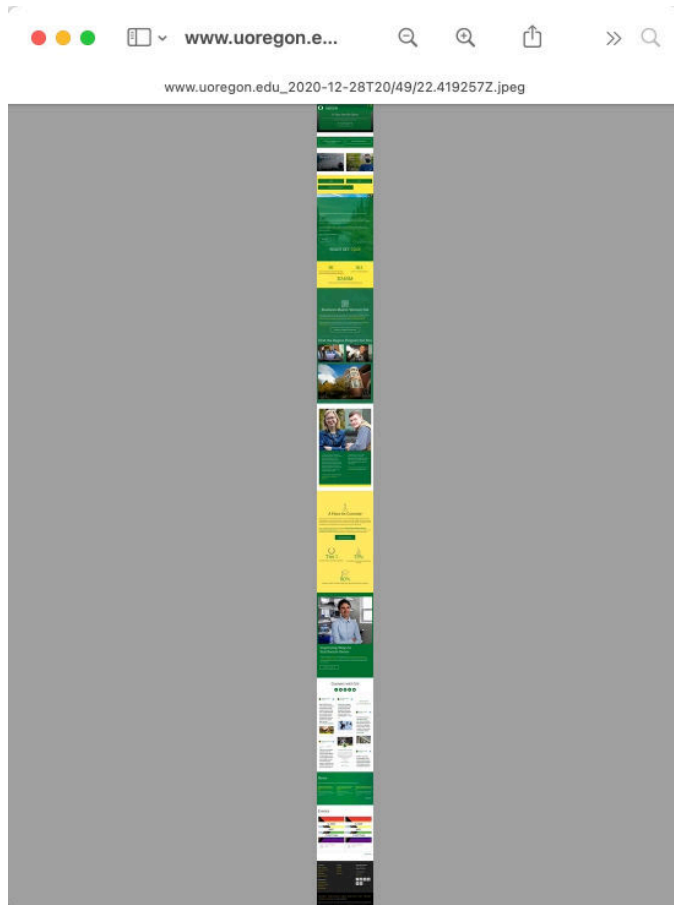
try:
    os.unlink(mylatest)
except:
    pass
os.symlink(myfilespec, mylatest)

```

- In Finder on macOS boxes, the captured file and its link gets displayed with slashes in place of colons. (The original filenames still show up if you check the file in Terminal)

Name	Size	Date Modified	Kind
www.uoregon.edu_2020-12-28T20/49/22.419257Z.jpeg	1.1 MB	Dec 28, 2020 at 12:49 PM	JPEG image
www.uoregon.edu_latest.jpeg	69 bytes	Dec 28, 2020 at 12:49 PM	Alias

- Looking at the scraped page capture in the macOS Preview.app, we see:



Those pages actually are all 800 pixels wide, but they appear tiny in that window because the image as a whole is actually 11,461 pixels in height. That extreme height means the image needs to be dynamically scaled in order to cram the whole thing into the small Preview window.

Clearly we'll need a **scrollable** display window in our application! The problem? Tkinter graphic windows are NOT scrollable. We'll explain how we'll deal with that process in the next section. Before going on to that section, a few additional notes about the screen grab process with Pypeteer:

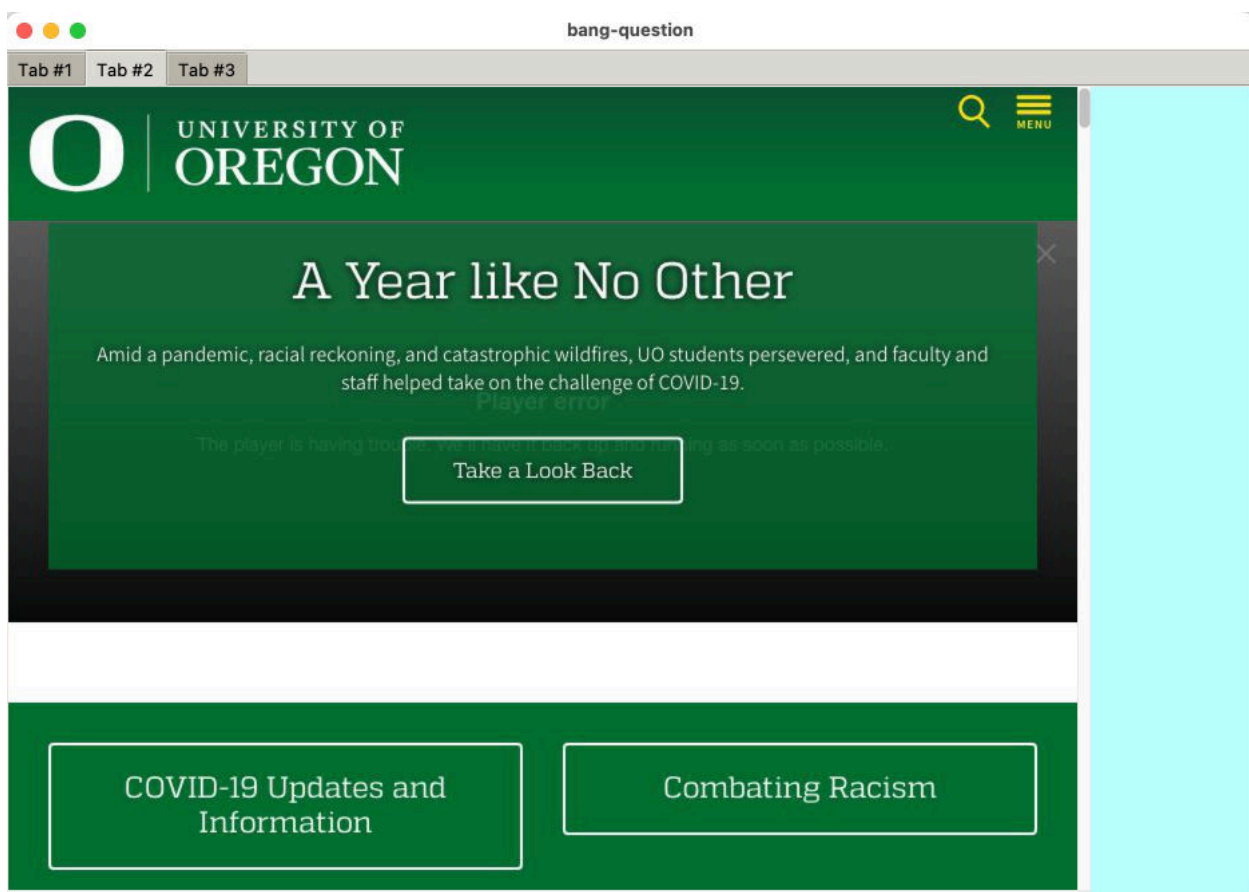
- While constructing our path, we're just concatenating the various parts of the filespec together with fixed Un*x-style file formatting. If we were prioritizing portability for our application, we'd probably use `os.path.join()` (instead of just concatenating elements) in order to get OS-aware path semantics, etc.
- In testing the sites we tried to screenshot, we only check port 443 and port 80. In some cases, sites may intentionally use non-standard ports (e.g., ports other than 443 and 80). We don't accommodate non-standard ports, nor URLs that include subdirectories. In a production application, rather than just a proof-of-concept application, those would be natural features to support.
- While grabbing sites, we make not attempt to "be sneaky." That is, we don't actively try to "look like"/"act like" a routine user on a run-of-the-mill web browser, nor do we route our connections through a proxy. It is thus entirely possible that our visit may be detectable by a defensively-alert webmaster/site admin. If that happens, the IP address you're working from may be discovered by the bad guys/bad gals.

10. DISPLAYING A SCROLLABLE JPEG IN TKINTER

Once we've captured our image with Pypeteer, we still need to display it in Tkinter. Two issues are potentially relevant:

- Mind bogglingly, Tkinter doesn't natively handle some common image formats (such as JPEG). Using the Python Imaging Library (Pillow) is the standard solution to overcoming this limitation.
- Tkinter also doesn't automatically handle scrolling if an image is oversized. We'll use a solution provided in <https://stackoverflow.com/questions/56043767/show-large-image-using-scrollbar-in-python/56043976> to overcome this deficiency.

The resulting output (for www.uoregon.edu) in Tab #2 looks like:



Note the scrollbar -- we can go through the full page by using the scrollbar to the right of the image, or by using two fingers on the Mac touchpad to do "mouse wheel" scrolling.

To see the code for all this, look at paragraph 12.

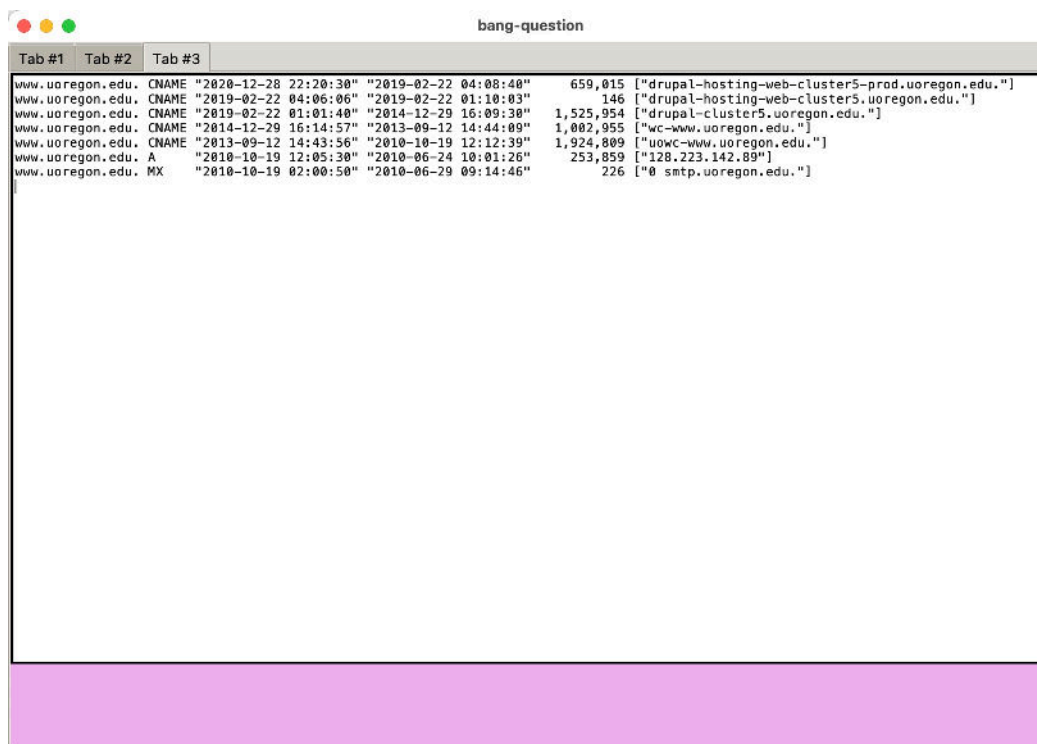
11. FILLING IN PASSIVE DNS RRNAME DATA ON OUR THIRD SAMPLE TAB

We'll now create a function in a separate file to grab DNSDB¹² RRname ("left hand side") data. We'll put that data into a scrollable text window for our third sample tab. Assumptions:

- There's a valid DNSDB API key in .dnsdb-apikey.txt in our home directory (note the leading dot!).
- You're fine using the public DNSDB server at api.dnsdb.info (rather than a local export installation).
- You only want/need the RRname, RRtype, time last seen, time first seen, and Rdata.
- You want to sort the results by time last seen (in descending order, e.g., most recent first).
- We don't care about (and don't want to be bothered with viewing) SOA records

We'll use pycurl¹³ to actually retrieve the data from the DNSDB API endpoint. The actual code for this can be seen in paragraph 12.

The output for Tab #3 now looks like:



```

www.uoregon.edu. CNAME "2020-12-28 22:20:30" "2019-02-22 04:08:40" 659,015 ["drupal-hosting-web-cluster5-prod.uoregon.edu."]
www.uoregon.edu. CNAME "2019-02-22 04:06:06" "2019-02-22 01:10:03" 146 ["drupal-hosting-web-cluster5.uoregon.edu."]
www.uoregon.edu. CNAME "2019-02-22 01:01:40" "2014-12-29 16:09:30" 1,525,954 ["drupal-cluster5.uoregon.edu."]
www.uoregon.edu. CNAME "2014-12-29 16:14:57" "2013-09-12 14:44:09" 1,002,955 ["wc-www.uoregon.edu."]
www.uoregon.edu. CNAME "2013-09-12 14:43:56" "2010-10-19 12:12:39" 1,924,809 ["uowc-www.uoregon.edu."]
www.uoregon.edu. A "2010-10-19 12:05:30" "2010-06-24 10:01:26" 253,859 ["128.223.142.89"]
www.uoregon.edu. MX "2010-10-19 02:00:50" "2010-06-29 09:14:46" 226 ["0 smtp.uoregon.edu."]

```

¹² <https://docs.dnsdb.info/>

¹³ <http://pycurl.io/>

12. A QUICK CHECKPOINT BEFORE WE GO ON...

You've now seen an example of how we can create a little sample app with three tabs -- one main tab with a menubar and log window, a second tab with a scraped copy of a web page we're interested in, and a third tab with some DNSDB RRname results. The full code for this example looks like the following three files runs about six pages all-in-all:

```
$ cat test-notebook-2.py
#!/usr/local/bin/python3

import asyncio
import sys

from tkinter import *
import tkinter as tk
import tkinter.ttk as ttk

import PIL
from PIL import ImageTk, Image, ImageDraw, ImageFont

from scrapePage import scrapeAFQDN          ### tab 2
from dnsdbRun import doRRnameQuery        ### tab 3

global FQDN, fqdn, t2, t3

class ScrollableImage(tk.Frame):
    def __init__(self, master=None, **kw):
        self.image = kw.pop('image', None)
        sw = kw.pop('scrollbarwidth', 10)
        super(ScrollableImage, self).__init__(master=master, **kw)
        self.cnvs = tk.Canvas(self, highlightthickness=0, **kw)
        self.cnvs.create_image(0, 0, anchor='nw', image=self.image)
        # Vertical and Horizontal scrollbars
        self.v_scroll = tk.Scrollbar(self, orient='vertical', width=sw)
        self.h_scroll = tk.Scrollbar(self, orient='horizontal', width=sw)
        # Grid and configure weight.
        self.cnvs.grid(row=0, column=0, sticky='nsew')
        self.h_scroll.grid(row=1, column=0, sticky='ew')
        self.v_scroll.grid(row=0, column=1, sticky='ns')
        self.rowconfigure(0, weight=1)
        self.columnconfigure(0, weight=1)
        # Set the scrollbars to the canvas
        self.cnvs.config(xscrollcommand=self.h_scroll.set,
                        yscrollcommand=self.v_scroll.set)
        # Set canvas view to the scrollbars
        self.v_scroll.config(command=self.cnvs.yview)
        self.h_scroll.config(command=self.cnvs.xview)
        # Assign the region to be scrolled
        self.cnvs.config(scrollregion=self.cnvs.bbox('all'))
        self.cnvs.bind_class(self.cnvs, "<MouseWheel>", self.mouse_scroll)

    def mouse_scroll(self, evt):
        if evt.state == 0 :
            self.cnvs.yview_scroll(-1*(evt.delta), 'units') # For MacOS
            self.cnvs.yview_scroll(int(-1*(evt.delta/120)), 'units') # For windows
        if evt.state == 1:
            self.cnvs.xview_scroll(-1*(evt.delta), 'units') # For MacOS
```

```

        self.cnvs.xview_scroll(int(-1*(evt.delta/120)), 'units') # For windows
def funca(event=None):
    global FQDN, fqdn, t2, t3
    fqdn = FQDN.get()
    print ("In funca, fqdn="+fqdn)

    mypage = asyncio.get_event_loop().run_until_complete\
        (scrapeAFQDN(fqdn, my_log_box))

    img2 = ImageTk.PhotoImage(Image.open(mypage))
    image_window = ScrollableImage(t2, image=img2,
        width=800, height=600)
    image_window.pack(anchor=NW)
    image_window.update()

    my_fqdn_results_json = doRRnameQuery(fqdn)

    # Text Widget height and width are in characters
    mytext_widget_pdns = tk.Text(t3,height=40,width=132)
    mytext_widget_pdns.pack(side=TOP, anchor=NW)
    mytext_widget_pdns.update()

    mytext_widget_pdns.insert(tk.END, my_fqdn_results_json)

def funcb(event=None):
    global FQDN
    print ("In funcb")
    FQDN.delete(0, 'end')

def funcc(event=None):
    sys.exit()

root = tk.Tk()
root.title("bang_question")
root.tk.call('wm','iconphoto',root._w,PhotoImage(file='exclamationquestion.gif'))

s = ttk.Style()
s.configure('TNotebook', tabposition='nw')
# to see a full list of potential themes: print(s.theme_names())
s.theme_use('clam')

mywindow = tk.Frame(root)
mynotebook = ttk.Notebook(mywindow)

# colors: www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter
t1 = tk.Frame(height=200,width=200,background='light goldenrod')
t2 = tk.Frame(height=200, width=200,background='PaleTurquoise1')
t3 = tk.Frame(height=200, width=200,background='plum2')

mymenubarbox = ttk.Frame(t1)
mymenubarbox.pack(side=TOP, anchor=NW)

mymenubarlabel = ttk.Label(mymenubarbox, text="    Enter FQDN:")
mymenubarlabel.pack(side=LEFT)

FQDN = ttk.Entry(mymenubarbox)
FQDN.pack(side=LEFT)

buttona = tk.Button(mymenubarbox, text='Submit', command = funca)

```

```

buttona.pack(side=LEFT)
root.bind('<Return>', funca)

buttonb = tk.Button(mymenubarbox, text='Clear', command = funcb)
buttonb.pack(side=LEFT)

buttonc = tk.Button(mymenubarbox, text='Quit', command = funcc)
buttonc.pack(side=LEFT)

mymenubarbox.pack(side=TOP, anchor=NW)
global my_log_box
my_log_box = tk.Text(t1,height=40,width=132)
my_log_box.pack(side=TOP, anchor=NW)

mynotebook.add(t1, text='Tab #1')
mynotebook.add(t2, text='Tab #2')
mynotebook.add(t3, text='Tab #3')

mynotebook.pack(expand=True, side=TOP, anchor=NW)
mywindow.pack(side=TOP, anchor=NW)

my_log_box.insert(tk.END, "Ready to run...\n")
my_log_box.update()

mywindow.mainloop()

```

```
$ cat scrapePage.py
```

```
#!/usr/local/bin/python3
```

```

import asyncio
import datetime
import errno
import os
from pathlib import Path
import socket

# https://pypi.org/project/pyppeteer2/
# https://miyakogi.github.io/pyppeteer/reference.html
# One time, download Chromium: $ pyppeteer-install
from pyppeteer import *

import tkinter as tk
from tkinter.messagebox import *

def display_error():
    showerror(message=\
        "Couldn't get FQDN.\nTypo in the FQDN entered?",
        title="Message Box", icon="error")

async def scrapeAFQDN(fqdn, my_log_box):
    # the snapshot goes to this filespec
    # if on something non-Un*x-ish, remember os.path.join(dir, f)
    my_log_box.insert(tk.END, "Making sure archive directory exists...\n")
    my_log_box.update()
    home = str(Path.home())
    subdir = "snapshots"
    myfilename = fqdn
    utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'
    myextension = "jpeg"

```

```

mydir = home + "/" + subdir
fullfilenamepart = fqdn + "_" + utcdate + "." + myextension
latestversion = fqdn + "_" + "latest" + "." + myextension
# ensure the directory exists
try:
    os.makedirs(mydir)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

# ensure the timestamped file exists
myfilespec = mydir + "/" + fullfilenamepart
try:
    open(myfilespec, 'a').close()
except:
    raise

my_log_box.insert(tk.END, "Confirming page is reachable on 443 or 80...\n")
my_log_box.update()

# verify page exists, and figure out if https is supported
try:
    # note: you do NOT need to explicitly permit inbound traffic on macOS!
    myaddrinfo = socket.getaddrinfo(fqdn, 443)
    url = "https://" + fqdn
    domain_resolves_ok = True
except socket.gaierror:
    # couldn't connect on default SSL/TLS port; fall back to regular http
    try:
        myaddrinfo = socket.getaddrinfo(fqdn, 80)
        url = "http://" + fqdn
        domain_resolves_ok = True
    except:
        # couldn't even connect on standard port
        display_error()
        return(1)

if domain_resolves_ok == True:
    browser = await launch({'headless': True, 'ignoreHTTPSErrors': True, \
        'defaultViewport': None, 'viewport_width': 800})
    context = await browser.createIncognitoBrowserContext()
    page = await browser.newPage()
    await page._client.send('Animation.disable')
    await page.goto(url)
    await page.screenshot({'path': myfilespec, 'type': 'jpeg', \
        'quality': 80, 'fullPage': True})
    await browser.close()
    # should we get and save cookies from the page, too?

    # ensure the "latest" version is updated for this URL
    mylatest = mydir + "/" + latestversion
    # print("mylatest =" + mylatest)

    try:
        os.unlink(mylatest)
    except:
        pass
    os.symlink(myfilespec, mylatest)

```

```
return(myfilespec)
```

```
$ cat dnssdbRun.py
#!/usr/local/bin/python3
from tkinter import *
from tkinter import scrolledtext
from pathlib import Path
from io import BytesIO
import pycurl
import json
from time import strftime, gmtime

# See stackoverflow.com/questions/26924812/python-sort-list-of-json-by-value
def extract_time(myrecord):
    json_format=eval(myrecord)

    try:
        extracted_bit = json_format['obj']['time_last']
    except:
        extracted_bit = json_format['obj']['zone_time_last']

    return extracted_bit

def doRRnameQuery(fqdn):
    content = make_query(fqdn)

    try:
        test = int(content)
        print("Error making dnssdb query! Return code = "+str(test))
        sys.exit(0)
    except:
        sList = list(line for line in content.strip().split("\n"))

        # we want to dump the first line in that output
        # print ("sList[0]="+sList[0])
        if sList[0] == '{"cond":"begin"}':
            sList.pop(0)
        else:
            print("SOMETHING ODD HAPPENED POPPING THE FIRST ELEMENT")

        # print ("sList[-1]="+sList[-1])
        if ((sList[-1] == '{"cond":"succeeded"}') or
            (sList[-1] == '{"cond":"limited","msg":"Result limit reached"}')):
            sList.pop()
        else:
            print("SOMETHING ODD HAPPENED POPPING THE LAST ELEMENT")

        sList2 = sorted(sList, key=extract_time, reverse=True)

        formatted_output=""
        results=""
        for line in sList2:
            results=print_bits(line)
            if results != "":
                result_with_nl=results+"\n"
                formatted_output=formatted_output+result_with_nl
```

```

        if len(formatted_output) == 0:
            formatted_output = "No results found\n"

        return(formatted_output)

def make_query(fqdn):
    # get the DNSDB API key
    filepath = str(Path.home()) + "/.dnsdb-apikey.txt"
    with open(filepath) as stream:
        myapikey = stream.read().rstrip()

    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rrset/name/" + fqdn

    requestHeader = []
    requestHeader.append('X-API-Key: ' + myapikey)
    requestHeader.append('Accept: application/jsonl')

    buffer = BytesIO()
    c = pycurl.Curl()
    c.setopt(pycurl.URL, url)
    c.setopt(pycurl.HTTPHEADER, requestHeader)
    c.setopt(pycurl.WRITEDATA, buffer)
    c.perform()
    rc = c.getinfo(c.RESPONSE_CODE)
    body = buffer.getvalue()
    content = body.decode('iso-8859-1')

    if rc == 200:
        return content
    else:
        return rc

def print_bits(myrecord):
    myformat = '%Y-%m-%d %H:%M:%S'
    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)

    try:
        extract_tl = myrecord_json_format['obj']['time_last']
    except:
        extract_tl = myrecord_json_format['obj']['zone_time_last']

    tl_datetime = gmtime(extract_tl)
    enddatetime = strftime(myformat, tl_datetime)

    try:
        extract_tf = myrecord_json_format['obj']['time_first']
    except:
        extract_tf = myrecord_json_format['obj']['zone_time_first']

    tf_datetime = gmtime(extract_tf)
    startdatetime = strftime(myformat, tf_datetime)

```



```

extract_count = myrecord_json_format['obj']['count']
formatted_count = str('{:>11,d}'.format(extract_count))
results = extract_bit + " " + extract_bit_2 + "\"\" + enddatetime + \"
    \" \" + startdatetime + \"\" \" + formatted_count + \"
    \" \" + extract_bit_3

if (results.find("SOA") == -1):
    return results
else:
    return ""

```

That's it for our little "colorful" test application.

What ***haven't*** we done yet? Well, obviously, there are still seven more tabs we still need to handle:

1. A DNSDB Rdata Search for FQDN --> IP
2. A check of the Domain WHOIS for the FQDN
3. A check of the IP WHOIS for the IP the FQDN is currently using
4. A check of the ASN WHOIS for ASN announcing the IP that the FQDN is currently using
5. A list of the prefixes associated with that ASN
6. A map showing the location associated with the IP address the FQDN is currently using
7. A network graph showing the DNSDB RRname search results

We'll get those remaining tabs out of the way in Part II.

We also need to deal with handling multiple sequential runs. If you currently try making multiple runs within the static three tab app we just fleshed out, you'll see that there's a "slight" problem: the content from the previous run doesn't get replaced by the content from the next run. The content from the 2nd, 3rd, ... Nth runs just gets ***added below the existing content***.

For example, consider tab #2 output from a run for `www.uoregon.edu` followed by a run for `www.washington.edu` as shown on the next page:

The screenshot shows a web browser window titled "bang-question" with three tabs (Tab #1, Tab #2, Tab #3). The main content area features a green header for the University of Oregon with a search icon and a menu icon. Below the header is a video player with the title "A Year like No Other" and a description: "Amid a pandemic, racial reckoning, and catastrophic wildfires, UO students persevered, and faculty and staff helped take on the challenge of COVID-19." The video player displays a "Player error" message: "The player is having trouble. We'll have it back up and running as soon as possible." A "Take a Look Back" button is visible below the error message. Below the video player is a green banner with two buttons: "COVID-19 Updates and Information" and "Combating Racism". At the bottom of the page is a purple footer for the University of Washington, featuring a large "W" logo, the text "UNIVERSITY of WASHINGTON", a search icon, and a "Quick Links" button with a right arrow.

That's NOT what we intuitively expected or wanted! We would have expected the University of Washington content to OVERWRITE the University of Oregon content, not just be appended below it.

Similarly, looking at tab #3, we see:

```

bang-question
Tab #1  Tab #2  Tab #3
www.uoregon.edu. CNAME "2021-01-01 10:41:17" "2019-02-22 04:08:40" 665,653 ["drupal-hosting-web-cluster5-prod.uoregon.edu."]
www.uoregon.edu. CNAME "2019-02-22 04:06:06" "2019-02-22 01:10:03" 146 ["drupal-hosting-web-cluster5.uoregon.edu."]
www.uoregon.edu. CNAME "2019-02-22 01:01:40" "2014-12-29 16:09:30" 1,525,954 ["drupal-cluster5.uoregon.edu."]
www.uoregon.edu. CNAME "2014-12-29 16:14:57" "2013-09-12 14:44:09" 1,002,955 ["wc-www.uoregon.edu."]
www.uoregon.edu. CNAME "2013-09-12 14:43:56" "2010-10-19 12:12:39" 1,924,809 ["uowc-www.uoregon.edu."]
www.uoregon.edu. A "2010-10-19 12:05:30" "2010-06-24 10:01:26" 253,859 ["128.223.142.89"]
www.uoregon.edu. MX "2010-10-19 02:00:50" "2010-06-29 09:14:46" 226 ["0 smtp.uoregon.edu."]

www.washington.edu. NS "2021-01-01 22:42:06" "2013-05-02 22:22:50" 37,951,562 ["dnsload11.s.uw.edu.", "dnsload12.s.uw.edu.", "dn
sload13.s.uw.edu."]
www.washington.edu. NS "2021-01-01 14:13:12" "2013-05-03 09:24:48" 3,227,938 ["dnsload11.s.uw.edu.", "dnsload12.s.uw.edu.", "dn
sload13.s.uw.edu."]
www.washington.edu. A "2021-01-01 11:55:04" "2012-05-03 19:14:05" 11,669,978 ["128.95.155.134", "128.95.155.197", "128.95.155.1
98"]
www.washington.edu. A "2021-01-01 11:48:04" "2012-05-03 19:14:22" 10,973,169 ["128.95.155.135", "128.95.155.197", "128.95.155.1
98"]
www.washington.edu. A "2021-01-01 11:45:49" "2012-05-03 19:14:15" 11,201,165 ["128.95.155.134", "128.95.155.135", "128.95.155.1
97"]
www.washington.edu. A "2021-01-01 11:38:56" "2012-05-03 19:16:28" 11,316,421 ["128.95.155.134", "128.95.155.135", "128.95.155.1
98"]
www.washington.edu. TXT "2020-12-14 05:29:44" "2012-05-05 19:47:58" 4,969 ["\"Contact: ds-apps@cac.washington.edu\""]
www.washington.edu. A "2020-07-28 07:21:56" "2019-06-20 20:07:38" 202 ["54.214.77.106"]

```

Again, having the new content appended to the bottom of the screen, BELOW the existing content, is not what we want.

In the next part, we'll recode the application so that the old content get destroyed before new content gets added to the notebook tabs.

Oh yes: our Clear button should also zap that content (unfortunately all it does currently is clear the domain name entry box).

We'll get all this fixed up in part II.

PART II.
FULL PROOF-OF-CONCEPT APP
WITH DYNAMIC TABS

13. TAB MANAGEMENT AND HANDLING MULTIPLE RUNS

We begin our rebuild by dealing with our erroneous "appending" (rather than "overwriting") of tab content. We'll also deal with our failure to properly clear old content from the tabs when the Clear button is pressed.

Conceptually, our **main tab, tab #1**, is the only tab we need to initially create (we need to automatically create it so we can enter the domain name of interest and get status reports on what's being done on our behalf). This means that our widget creation code (excluding imports, global statements, etc.) in the **main routine** now looks like:

```

root = tk.Tk()
root.title("bang_question")
iconfilespec = str(Path.home()) + "/.bang_question_files/" + \
    "exclamationquestion.gif"
root.tk.call('wm', 'iconphoto', root._w, ImageTk.PhotoImage(file=iconfilespec))
s = ttk.Style()
s.configure('TNotebook', tabposition='nw')
s.theme_use('clam')
root.bind('<Return>', funca)
mywindow = tk.Frame(root)

# create notebook
mynotebook = ttk.Notebook(mywindow)
mynotebook.pack(side=TOP, anchor=NW)

t1 = tk.Frame()
mynotebook.add(t1, text="MAIN")

# build menu box
mymenubarbox = ttk.Frame(t1)
mymenubarbox.pack(side=TOP, anchor=NW)
mymenubarlabel = ttk.Label(mymenubarbox, text="    Enter FQDN:")
mymenubarlabel.pack(side=LEFT)
FQDN = ttk.Entry(mymenubarbox)
FQDN.pack(side=LEFT)
buttona = tk.Button(mymenubarbox, text='Submit', command = funca)
buttona.pack(side=LEFT)
buttonb = tk.Button(mymenubarbox, text='Clear', command = funcb)
buttonb.pack(side=LEFT)
buttonc = tk.Button(mymenubarbox, text='Quit', command = funcc)
buttonc.pack(side=LEFT)
mymenubarbox.pack(side=TOP, anchor=NW)

# build log box
my_log_box = tk.Text(t1,height=40,width=132)
my_log_box.pack(side=TOP, anchor=NW)

# add tab #1 to notebook
mynotebook.add(t1, text="MAIN")
mynotebook.pack(side=TOP, anchor=NW, expand=True)

mywindow.pack(side=TOP, anchor=NW)
mywindow.mainloop()

```

The rest of the tabs (tabs 2 through 10) we'll create in `funca` (the callback for the "Submit" button). **That function is our largest chunk of code.** Let's now look at the "guts" of `funca`, excluding logging statements, global statements, etc.

Starting with **tab #1 (our "main" tab)**, we're clearing the log box, and we're picking up the FQDN to analyze:

```
my_log_box.delete('1.0', END)
fqdn = FQDN.get()
```

After a FQDN is entered and our completed program runs, that tab will look like:

The screenshot shows a web application with a horizontal tab bar at the top containing the following tabs: MAIN, Screen Grab, DNSDB RRnames, DNSDB IP Rdata, DomWhois, IPWhois, ASNWhois, Prefixes, GeoIP, and Graph. The 'MAIN' tab is active. Below the tabs is an input field labeled 'Enter FQDN:' containing the text 'www.uoregon.edu'. To the right of the input field are three buttons: 'Submit', 'Clear', and 'Quit'. Below the input field is a large text area displaying the following log output:

```
New run initiated
The FQDN entered was www.uoregon.edu
About to do screen grab of FQDN...
Making sure archive directory exists...
Confirming page is reachable on 443 or 80...
Screen grab done. Grabbed page is
/Users/joe/snapshots/www.uoregon.edu_2021-01-07T00:36:50.679857Z.jpeg
Screen grab added to tab.
Running DNSDB RRname query now...
Passive DNS results added to tab.
Running DNSDB Rdata IP query now...
Passive DNS Rdata IP results added to tab.
Now working on Domain WHOIS...
Domain WHOIS results added to tab.
Now working on IP WHOIS...
IP WHOIS results added to tab.
Now working on ASN WHOIS results...
Making sure archive directory exists...
Confirming page is reachable on 443 or 80...
ASN Whois info added to tab.
Now working on Adding prefixes...
ASN Prefixes added to tab.
Now working on geolocation..
Geolocation map added to tab.
Working on network graph...
Network graph added to tab.
--DONE--
```

In **tab #2 (our "Screen Grab" tab)**, we'll begin by doing the screen grab we want to display:

```
mypage = asyncio.get_event_loop().run_until_complete\
(scrapeAFQDN(fqdn, my_log_box))
```

We'll then try to destroy the `t2` tab (if a `t2` tab already exists):

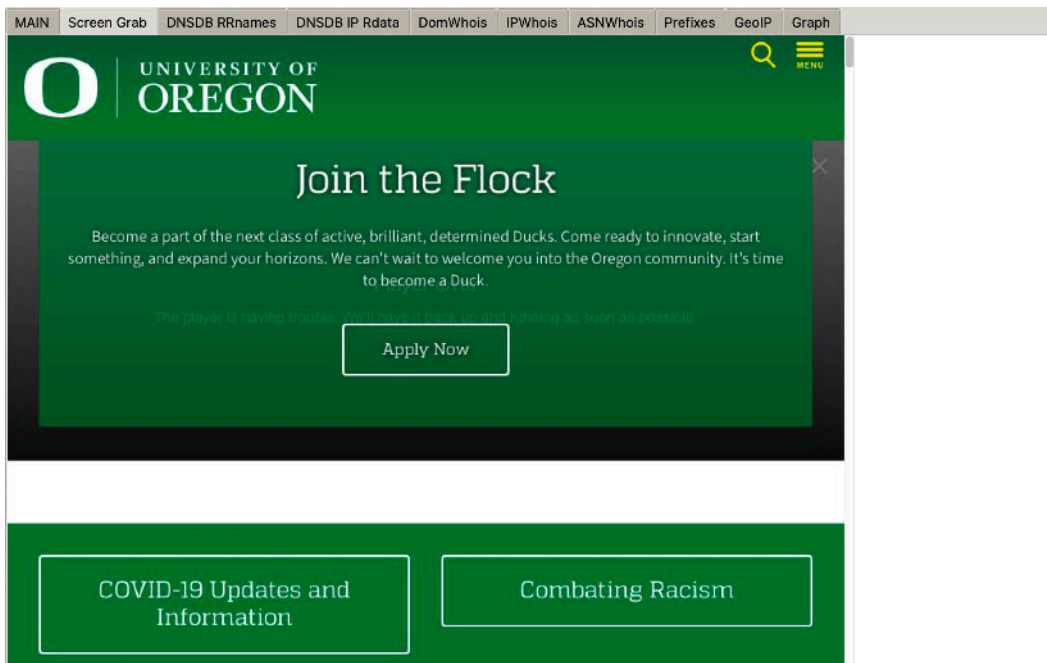
```
try:
    t2.destroy()
except:
```

pass

Then we'll (re)-create the frame, add it to our notebook, populate it with our image, pack the widget, and ensure it gets promptly displayed by calling update:

```
t2 = tk.Frame()
mynotebook.add(t2, text="Screen Grab")
img2 = ImageTk.PhotoImage(Image.open(mypage))
image_window = ScrollableImage(t2, image=img2,
    width=800, height=600)
image_window.pack(anchor=NW)
image_window.update()
```

When that code runs, that tab will look like:



In **tab #3** (our "**DNSDB RRnames**" tab), we'll do the first of three DNSDB queries. We consolidated all the DNSDB query types into a single integrated "doQuery" function, and depending on the 2nd argument passed to that function, one of three query types will be run:

- "full" ==> RRname query with output of RRname, RRtype, TimeFirst, TimeLast, Rdata and Count
- "limited" ==> RRname query with just output of RRname, RRtype and Rdata (this is for graphing)
- "RdataIP" ==> Rdata IP address query

For the purposes of tab #3, we want to do a "full" RRname query:

```
my_fqdn_results_json = doQuery(fqdn, "full")
```

We then add the results to the tab:

```

try:
    t3.destroy()
except:
    pass

t3 = tk.Frame()
mynotebook.add(t3, text="DNSDB RRnames")

# Text Widget height and width are in characters
mytext_widget_pdns = tk.Text(t3,height=40,width=132)
mytext_widget_pdns.pack(side=TOP, anchor=NW)
mytext_widget_pdns.update()

```

We'll handle the other seven tabs the same way. When run, this third tab will look like:

MAIN	Screen Grab	DNSDB RRnames	DNSDB IP Rdata	DomWhois	IPWhois	ASNWhois	Prefixes	GeoIP	Graph
www.uoregon.edu.	CNAME	"2021-01-06 15:10:57"	"2019-02-22 04:08:40"		677,845	["drupal-hosting-web-cluster5-prod.uoregon.edu."]			
www.uoregon.edu.	CNAME	"2019-02-22 04:06:06"	"2019-02-22 01:10:03"		146	["drupal-hosting-web-cluster5.uoregon.edu."]			
www.uoregon.edu.	CNAME	"2019-02-22 01:01:40"	"2014-12-29 16:09:30"		1,525,954	["drupal-cluster5.uoregon.edu."]			
www.uoregon.edu.	CNAME	"2014-12-29 16:14:57"	"2013-09-12 14:44:09"		1,802,955	["uc-www.uoregon.edu."]			
www.uoregon.edu.	CNAME	"2013-09-12 14:43:56"	"2010-10-19 12:12:39"		1,924,809	["uowe-www.uoregon.edu."]			
www.uoregon.edu.	A	"2010-10-19 12:06:30"	"2010-06-24 10:01:26"		253,859	["128.223.142.89"]			
www.uoregon.edu.	MX	"2010-10-19 02:00:50"	"2010-06-29 09:14:46"		226	["0 smtp.uoregon.edu."]			

For example, **tab #4** (our "DNSDB RdataIP" tab) is quite similar, except we pass myip (instead of fqdn) and tag this as an "RdataIP" query:

```

myip = socket.gethostbyname(fqdn)
my_ip_results_json = doQuery(myip, "RdataIP")

try:
    t4.destroy()
except:
    pass

t4 = tk.Frame()
mynotebook.add(t4, text="DNSDB IP Rdata")

# Text Widget height and width are in characters
mytext_widget_pdns_2 = tk.Text(t4,height=40,width=132)
mytext_widget_pdns_2.pack(side=TOP, anchor=NW)

```



```
mytext_widget_pdns_2.update()
```

When run, this tab will look like:

MAIN	Screen Grab	DNSDB RRnames	DNSDB IP Rdata	DomWhois	IPWhois	ASNWhois	Prefixes	GeoIP	Graph
		universityoforegon.education.	A	*2021-01-06 16:04:06*	*2019-04-24 02:58:48*			411	["184.171.111.233*"]
		drupal-hosting-web-cluster5-prod.uoregon.edu.	A	*2021-01-06 14:27:48*	*2019-02-04 00:29:38*			593,845	["184.171.111.233*"]
		uoregon.edu.	A	*2021-01-06 13:15:29*	*2019-02-22 01:10:03*			336,974	["184.171.111.233*"]
		uoregon.us.	A	*2021-01-05 15:55:57*	*2019-04-19 16:12:14*			360	["184.171.111.233*"]
		uoregon.biz.	A	*2021-01-03 23:31:08*	*2019-04-21 00:59:11*			449	["184.171.111.233*"]
		uoregon.me.	A	*2021-01-01 15:27:36*	*2019-04-19 03:41:18*			259	["184.171.111.233*"]
		uoregon.edu.1.1.7b52bd64.roksit.net.	A	*2019-07-30 08:34:02*	*2019-07-30 08:34:02*			2	["184.171.111.233*"]
		drupal-hosting-web-cluster5.uoregon.edu.	A	*2019-02-22 04:06:06*	*2019-02-22 01:10:03*			154	["184.171.111.233*"]

Now we do **tab #5**, our **Domain Whois** tab. Performing this query is complicated by the fact that there are now over 1,200 different ICANN-recognized TLDs,¹⁴ each TLD can have a different authoritative WHOIS server (or none at all), and domains may now include "internationalized" domains with non-ASCII characters.

Ideally, what we'd like would be to programmatically get something akin to what we'd get from a command line Whois client (such as Marco D'Itri's excellent command line Whois client¹⁵). Here's an example of what text output from that client looks like:

```
$ whois uoregon.edu
[...]
```

```
Domain Name: UOREGON.EDU

Registrant:
  University of Oregon
  1225 Kincaid St
  Eugene, OR 97403
  US
```

```
Administrative Contact:
  UO Domain Name Management
  University of Oregon
```

¹⁴ <https://newgtlds.icann.org/en/program-status/statistics>

¹⁵ <https://github.com/rfc1036/whois>

1212 University of Oregon
 Eugene, OR 97403-1212
 US
 +1.541346
 uoregon-dns@uoregon.edu

Technical Contact:

UO Domain Name Management
 University of Oregon
 1212 University of Oregon
 Eugene, OR 97403-1212
 US
 +1.541346
 uoregon-dns@uoregon.edu

Name Servers:

NS1.F5CLOUDSERVICES.COM
 RUMINANT.UOREGON.EDU
 PHLOEM.UOREGON.EDU
 LSU-BDDS1.LSU.EDU

Domain record activated: 23-Feb-1988
 Domain record last updated: 26-Dec-2020
 Domain expires: 31-Jul-2021

Normally, for a bulletproof programmatic solution, we'd use a commercial Domain WHOIS API solution from someone who specializes in collecting and parsing Domain Whois such as DomainTools' Domain Whois API¹⁶ or iThreat's CyberTOOLBELT API.¹⁷ [Note: DomainTools and CyberToolbelt are Farsight Security partners.¹⁸]

For the purpose of this example, however, we wanted to see what we could do with a free option. We decided to try Danny Cork's Python Whois library.¹⁹ If we're using Danny Cork's Python Whois library, all we need to add to our Python code is a suitable import statement near the top:

```
# https://github.com/DannyCork/python-whois
import whois
```

And then in `funca`, we'll use:

```
domain = whois.query(fqdn)
domain_json = json.dumps(domain.__dict__, indent=4, default=str)

try:
    t5.destroy()
except:
    pass

t5 = tk.Frame()
mynotebook.add(t5, text="DomWhois")
mytext_widget_5 = tk.Text(t5,height=40,width=132)
mytext_widget_5.pack(side=TOP, anchor=NW)
```

¹⁶ <https://www.domaintools.com/resources/api-documentation/whois-lookup>

¹⁷ <https://www.cybertoolbelt.com/index.php#api>

¹⁸ See <https://www.farsightsecurity.com/partners/all/>

¹⁹ <https://github.com/DannyCork/python-whois>

```
mytext_widget_5.insert(tk.END, domain_json)
```

That said, the JSON format output from that library that looks *quite* different from what we get from a normal command-line WHOIS client -- in particular, we don't get any point-of-contact information:



```
{
  "name": "uoregon.edu",
  "registrar": "",
  "registrant_country": "",
  "creation_date": "1988-02-23 00:00:00",
  "expiration_date": "2021-07-31 00:00:00",
  "last_updated": "2020-12-26 00:00:00",
  "status": "",
  "statuses": [
    ""
  ],
  "dnssec": false,
  "name_servers": ["ns1.f5cloudservices.com", "phloem.uoregon.edu"]
}
```

This is nonetheless sufficient for proof-of-concept purposes.

Handling **tab #6 (IP Whois)** is very similar to tab #5. For the IP Whois Checks, we decided to use IPWhois.²⁰ Doing so involves adding the following line to the imports area of the code:

```
from ipwhois import IPWhois
```

All we need substantively in the funca area is:

```
myip = socket.gethostbyname(fqdn)
obj = IPWhois(myip)
res=obj.lookup_whois(get_referral=True)
pretty_printed_text = json.dumps(res, indent=4)

try:
    t6.destroy()
except:
    pass

t6 = tk.Frame()
mynotebook.add(t6, text="IPWhois")
mytext_widget_6 = tk.Text(t6,height=40,width=132)
mytext_widget_6.pack(side=TOP, anchor=NW)
mytext_widget_6.insert(tk.END, pretty_printed_text)
```

²⁰ <https://github.com/secynic/ipwhois>

Typical output looks like:

```

{
  "asn_registry": "arin",
  "asn": "3582",
  "asn_cidr": "184.171.0.0/17",
  "asn_country_code": "US",
  "asn_date": "2010-10-08",
  "asn_description": "UUNET, US",
  "query": "184.171.111.233",
  "nets": [
    {
      "cidr": "184.171.0.0/17",
      "name": "UUNET",
      "handle": "NET-184-171-0-1",
      "range": "184.171.0.0 - 184.171.127.255",
      "description": "University of Oregon",
      "country": "US",
      "state": "OR",
      "city": "Eugene",
      "address": "UO Information Services\n1225 Kincaid Street",
      "postal_code": "97403",
      "emails": [
        "uonethe1p@uoregon.edu",
        "jfo@uoregon.edu",
        "abuse@uoregon.edu",
        "dteach@uoregon.edu",
        "jad@uoregon.edu"
      ],
      "created": "2010-10-08",
      "updated": "2012-03-02"
    }
  ],
  "raw": null,
  "referral": null,
  "raw_referral": null
}

```

Now we'll do **tab #7, our ASN Whois tab**. Tab #7 is sufficiently complex that we've put most of it into the function `myAsnWhois`, which we'll describe in a subsequent section. For now, it is sufficient to understand that we'll be producing a local HTML file which we then "scrape" to create an image for display in the tab:

```

# scraping the ASN Whois
my_asn_info_file = myAsnWhois(fqdn)
myasnpage = asyncio.get_event_loop().run_until_complete\
    (scrapeAFQDN(my_asn_info_file, my_log_box, "file"))

try:
    t7.destroy()
except:
    pass

t7 = tk.Frame()
mynotebook.add(t7, text="ASNWhois")
asning7 = ImageTk.PhotoImage(Image.open(myasnpage))
image_window = ScrollableImage(t7, image=asning7,
    width=1000, height=600)
image_window.pack(anchor=NW)
image_window.update()

```

Sample output for this tab looks as shown below. Note that this tab is showing a screen shot of the HTML rendered page.

The screenshot shows the ARIN WHOIS-RWS interface. At the top, there are navigation tabs: MAIN, Screen Grab, DNSDB RRnames, DNSDB IP Rdata, DomWhois, IPWhois, ASNWhois, Prefixes, GeoIP, and Graph. The ARIN logo is on the left, and a search bar is on the right. The main content area is titled 'WHOIS-RWS' and displays the following information:

Autonomous System Number	
Number	3582
Name	UONET
Handle	AS3582
Organization	University of Oregon (UNIVER-193)
Registration Date	1994-05-11
Last Updated	2019-04-11
Comments	
RESTful Link	https://whois.arin.net/rest/asn/AS3582

Below this is a table for 'Point of Contact':

Function	Point of Contact
Tech	TEACH4-ARIN (TEACH4-ARIN)
Abuse	UAD4-ARIN (UAD4-ARIN)
NOC	UONET-ARIN (UONET-ARIN)

To the right of the main table is a 'RELEVANT LINKS' section with the following links:

- [ARIN Whois/Whois-RWS Terms of Service](#)
- [Report Whois Inaccuracy](#)
- [Search ARIN Whois with RDAP](#)

Tab #8 has a list of network prefixes announced by the ASN that announced the IP we're interested in.

To do tab #8, we go from FQDN --> IP(FQDN) --> ASN(IP(FQDN)) --> Prefixes announced by that ASN. In this case we look it up via an IP-to-ASN database that we found, but we could also have done a live query against the Routeviews IP-to-ASN service available via DNS:

```
# we need the ASN for some of the filenames, so we'll get that first
myip2asnfilespec = str(Path.home()) + "/.bang_question_files/" + \
    "my_ip2asn_db_file"
asndb = pyasn.pyasn(myip2asnfilespec)
asn = asndb.lookup(myip)
```

Once we have the ASN, the same database will let us get a report of the associated ASNs:

```
# get the prefixes associated with the ASN
prefixes = asndb.get_as_prefixes(asn[0])
prefixes = sorted(prefixes)
```

Because the prefixes are returned as a Python list, we put the list elements together into a single long text string separated with newlines. After that, things proceed normally for a text item:

```
combined_text=""
for pre in prefixes:
    combined_text=combined_text+"\n"+pre

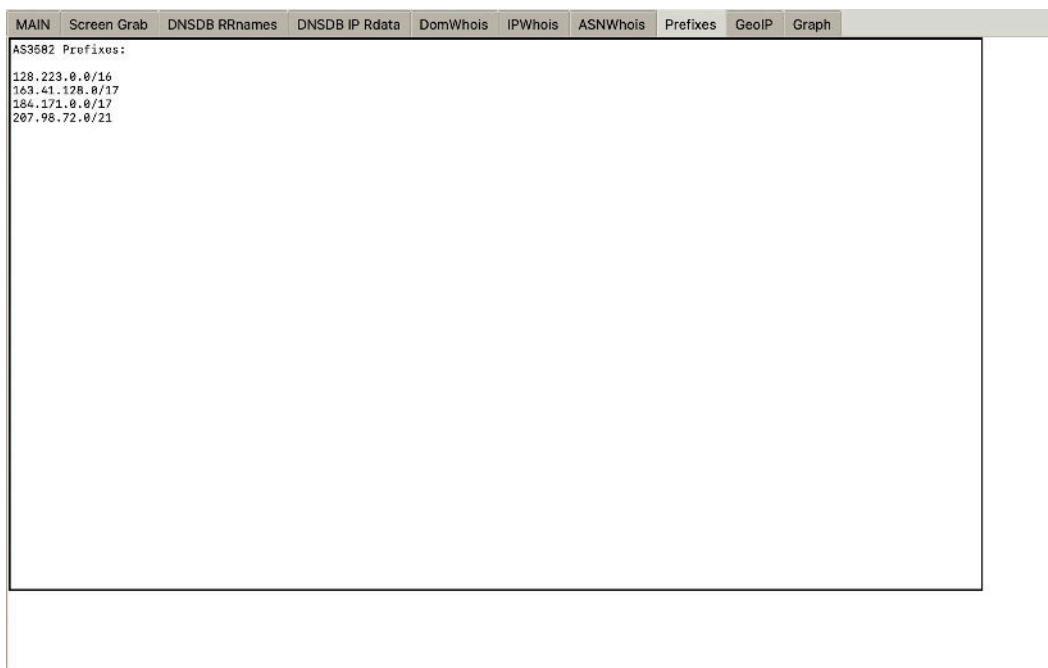
try:
    t8.destroy()
except:
    pass

t8 = tk.Frame()
mynotebook.add(t8, text="Prefixes")
mytext_widget_8_prefixes = tk.Text(t8,height=40,width=132)
```

```

mytext_widget_8_prefixes.pack(side=TOP, anchor=NW)
mytext_widget_8_prefixes.insert(tk.END, "AS"+str(asn[0])+" Prefixes:\n")
mytext_widget_8_prefixes.insert(tk.END, combined_text)

```



Tab #9 is the **geolocation tab**. The code for this tab is actually long enough we should probably have pulled it into its own stand-alone function. Nonetheless, just to mix it up, we'll leave the code inline.

We begin by setting up the filenames we need. We create two filenames, one with the full filename and another one that's a "latest" version of the maps for that FQDN:

```

home = str(Path.home())
subdir9 = "geolocation-maps"
myfilename9 = fqdn
utcdatetime9 = datetime.datetime.utcnow().isoformat() + 'Z'
myextension9 = "jpeg"
mymapextension9 = "png"
mydir9 = home + "/" + subdir9
fullfilenamepart9 = fqdn + "_" + utcdatetime9 + "." + myextension9
latestversion9 = fqdn + "_" + "latest" + "." + myextension9
mymapname9 = mydir9 + "/" + fqdn + "_" + utcdatetime9 + "." + \
    mymapextension9
myfilespec9 = mydir9 + "/" + fullfilenamepart9

```

After the first run, the directory will exist, but we'll still check and force its creation if it doesn't exist:

```

# ensure the directory exists
try:
    os.makedirs(mydir9)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

```

Now we make sure we can create the two files we need:

```

try:
    open(myfilespec9, 'a').close()
except:
    raise

try:
    open(mymapname9, 'a').close()
except:
    raise

```

Now we're ready to actually do the geolocation. We're using IP2Location-Lite, which uses a local database file for the look up:

```

geoipfilespec = str(Path.home()) + "/.bang_question_files/" + \
    "IP2LOCATION-LITE-DB9.IPV6.BIN"
database = IP2Location.IP2Location(geoipfilespec, "SHARED_MEMORY")
rec = database.get_all(myip)
lat = rec.latitude
lon = rec.longitude
cit = rec.city

```

Mapping the data with Plotly Express is easiest if the data is transformed into a Panda Dataframe first:

```

# map it (we need to make it into a DataFrame first
d={"txt":cit,"lat":lat,"lon":lon,"siz":10}
df=pd.DataFrame(d,columns=["txt","lat","lon","siz"])

```

We choose between two map projections ("albers usa" or "mercator"), depending on whether we're in the US (or southern Canada), which we handle as a special case, or anywhere else in the world:

```

if ((lon >= -180) and (lon <= -52) and
    (lat >= 36) and (lat <= 83)):
    fig=px.scatter_geo(df,lat="lat",lon="lon",text="txt",\
        size="siz",projection="albers usa",width=800,height=600)
else:
    fig=px.scatter_geo(df,lat="lat",lon="lon",text="txt",\
        size="siz",projection="mercator",width=800,height=600)

fig.write_image(mymapname9)

# ensure the image is appropriately sized
map_img=Image.open(mymapname9).resize((800, 600),Image.ANTIALIAS)
map_img.load()

```

Unfortunately, the image emitted by Plotly Express has an "alpha channel" (it's an RGBA file rather than RGB). We found a fix for this, and created a new RGB-only file:

```

# BUG: https://stackoverflow.com/questions/42099914/imagetk-photoimage-doesnt-show-
up-on-osx-but-does-on-windows
# See https://stackoverflow.com/questions/41576637/are-rgba-pngs-unsupported-in-
python-3-5-pillow
# See Yuji Tomita's post at https://stackoverflow.com/questions/9166400/convert-rgba-
png-to-rgb-with-pil
background2 = Image.new("RGB", map_img.size, (255,255,255))
background2.paste(map_img, mask=map_img.split()[3]) # 3 is the alpha channel

```

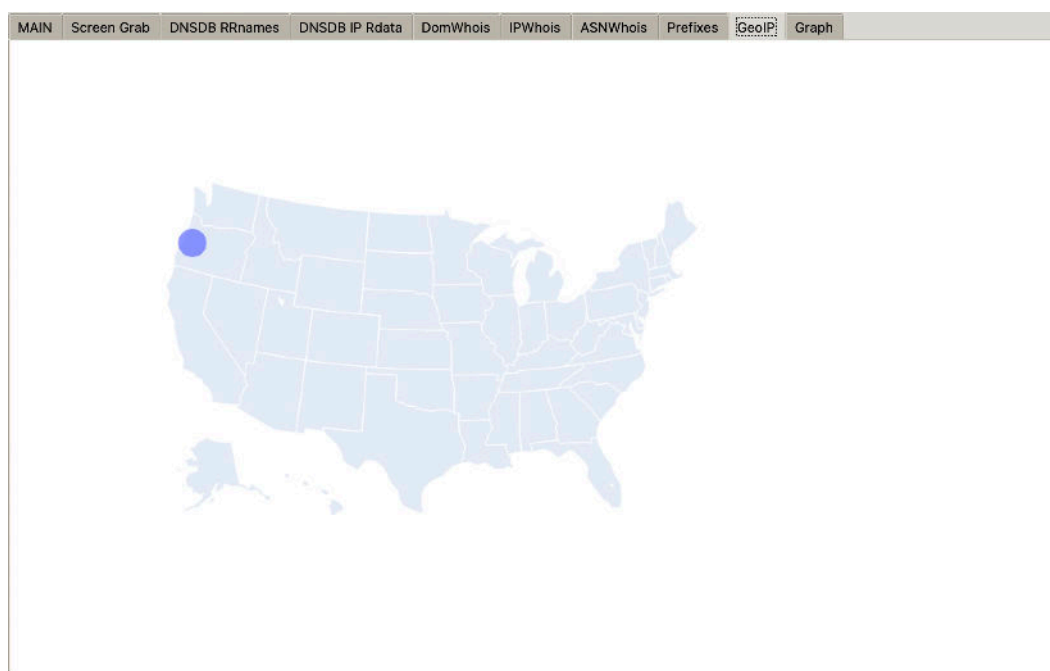
```
background2.save(myfilespec9, 'JPEG', quality=80)
img4 = ImageTk.PhotoImage(background2)
```

After that, everything's normal/typical:

```
try:
    t9.destroy()
except:
    pass

t9 = tk.Frame()
mynotebook.add(t9, text="GeoIP")
panel2 = tk.Label(t9, image=img4)
panel2.pack(side=TOP, anchor=NW)
mynotebook.add(t9)
```

Sample tab #9 output looks like:



And finally, we'll handle launching **tab #10, our network graph tab**. What's a little different about this tab is that the network graph is being created as a PDF file, and then read and converted with "Fitz."

We begin by setting up the filenames we need, much as we did in tab #9:

```
subdir10 = "network-graphs"
myfilename10 = fqdn
utcdatetime10 = datetime.datetime.utcnow().isoformat() + 'Z'
myextension10 = "pdf"
mydir10 = home + "/" + subdir9
fullfilenamepart10 = fqdn + "_" + utcdatetime10 + "." + myextension10
myfilespec10 = mydir10 + "/" + fullfilenamepart10
latestversion10 = fqdn + "_" + "latest" + "." + myextension10
```

Once again we ensure the directory and file exist:


```

# ensure the directory exists
try:
    os.makedirs(mydir10)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

try:
    open(myfilespec10, 'a').close()
except:
    raise

```

The actual process of creating the image is just a one liner, combining a DNSDB call to get the data with another function that will draw the graph:

```
draw_the_graph(doQuery(fqdn, "limited"), myfilespec10)
```

This is where we use fitz to convert the PDF to a byte-image:

```

doc=fitz.open(myfilespec10)
page=doc.loadPage(0)
pix=page.getPixmap()
mode="RGB"

```

Now we proceed fairly normally to handle this final tab:

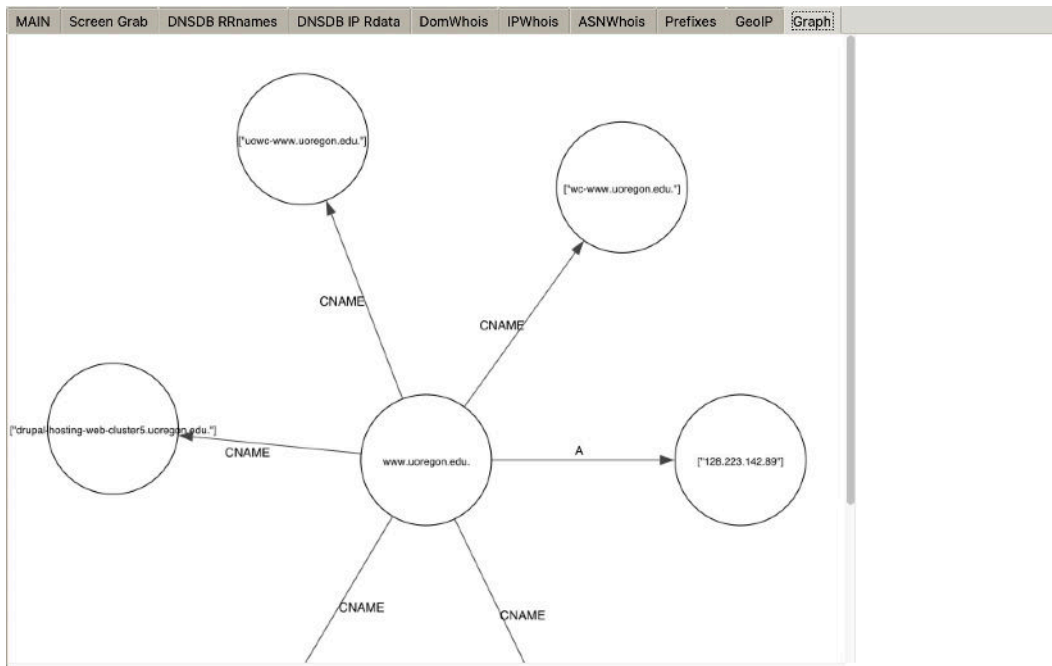
```

try:
    t10.destroy()
except:
    pass

t10 = tk.Frame()
mynotebook.add(t10, text="Graph")
t10img=Image.frombytes(mode, [pix.width, pix.height], pix.samples)
t10tkimg=ImageTk.PhotoImage(t10img)
t10_image_window = ScrollableImage(t10, image=t10tkimg,
    width=800, height=600)
t10_image_window.pack(anchor=NW)

```

A sample version of our final tab looks like:



"While we're here," so-to-speak, let's also handle the **"Clear" button** (aka **"funcb"**), and the **"Quit" button** (aka **"funcc"**).

```

def cleanup_tab(tab_to_cleanup):
    try:
        tab_to_cleanup.destroy()
    except:
        pass

def funcb(event=None):
    global FQDN, my_log_box
    global t1, t2, t3, t4, t5, t6, t7, t8, t9, t10

    FQDN.delete(0, 'end')

    my_log_box.delete('1.0', END)
    my_log_box.update()

    # we intentionally clean these up in reverse order and leave t1 alone
    my_tabs = [t10, t9, t8, t7, t6, t5, t4, t3, t2]
    for i in my_tabs:
        cleanup_tab(i)
  
```

The **Quit button callback, funcc**, remains pretty basic:

```

### Callback for quit button
def funcc(event=None):
    sys.exit()
  
```

If you look at the actual code (as shown in the Appendix II), you'll see we've added logging statements such as:

```
my_log_box.insert(tk.END, "New run initiated\n")
my_log_box.update()
```

Each log message will normally end with an explicit newline (backslash n). The tk.END argument ensures we add the new entry to the bottom of the log box. The my_log_box.update() statement ensures that the log message gets displayed immediately, rather than potentially being buffered/delayed.

14. SCRAPING A PAGE

So tab #2 gets a scraped copy of the FQDN the user specified. We'll go through the whole function that makes that happens. We include this function into the mainline routine with:

```
# make sure you don't include the .py as part of the name of the imports
from scrapePage import scrapeAFQDN
```

Here's the scrapePage.py file. We begin with our import statements, including both standard libraries and the key-to-screen-scraping Pypeteer library. We also define a simple error display routine to handle typo'd FQDNs:

```
$ cat scrapePage.py
import asyncio
import datetime
import errno
import os
from pathlib import Path
import socket

# https://pypi.org/project/pypeteer2/
# https://miyakogi.github.io/pypeteer/reference.html
# One time, download Chromium: $ pypeteer-install
from pypeteer import *

import tkinter as tk
from tkinter.messagebox import *

def display_error():
    tkinter.messagebox.showerror(message=\
        "Couldn't get FQDN.\nTypo in the FQDN entered?",
        title="Message Box", icon="error")
```

Our scrapeAFQDN function begins with building the filenames we need, and ensuring the directory and file exist. There's a slight twist to this because we could be scraping a remote web site (a "url") or a local html file:

```
async def scrapeAFQDN(fqdn, my_log_box, url_or_file):
    # the snapshot goes to this filespec
    # if on something non-Un*x-ish, remember os.path.join(dir, f)
    my_log_box.insert(tk.END, "Making sure archive directory exists...\n")
    my_log_box.update()
    home = str(Path.home())
    subdir = "snapshots"
    myfilename = fqdn
    utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'
```

```

myextension = "jpeg"
mydir = home + "/" + subdir
fullfilenamepart = fqdn + "_" + utcdatetime + "." + myextension
latestversion = fqdn + "_" + "latest" + "." + myextension

# ensure the directory exists
try:
    os.makedirs(mydir)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

# ensure the timestamped file exists
if (url_or_file == "url"):
    myfilespec = mydir + "/" + fullfilenamepart
    # ensure the directory exists
    try:
        os.makedirs(mydir)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise
elif (url_or_file == "file"):
    myfilespec = fullfilenamepart
try:
    open(myfilespec, 'a').close()
except:
    raise

```

The next major chunk of this function handles confirming that the resource to be scraped is reachable on one of the two ports we're going to try, either 443 (https) or 80 (http):

```

my_log_box.insert(tk.END, "Confirming page is reachable on 443 or 80...\n")
my_log_box.update()

domain_resolve_ok = False
if (url_or_file == "url"):
    # verify page exists, and figure out if https is supported
    try:
        # note: you do NOT need to explicitly permit inbound traffic
        # on macOS (even if it asks you to allow it!)
        myaddrinfo = socket.getaddrinfo(fqdn, 443)
        url = "https://" + fqdn
        domain_resolves_ok = True
    except socket.gaierror:
        # couldn't connect on default SSL/TLS port
        # fall back to regular http
        try:
            myaddrinfo = socket.getaddrinfo(fqdn, 80)
            url = "http://" + fqdn
            domain_resolves_ok = True
        except:
            # couldn't even connect on standard port
            display_error()
            return(1)
elif (url_or_file == "file"):
    domain_resolves_ok = True
    url = "file://" + fqdn
    myfilespec = fqdn + ".jpeg"

```

Assuming the site (or local html file) to be scraped is reachable, we actually scrape it. We're going to go through this bit line by line:

```
if domain_resolves_ok == True:
    browser = await launch({'headless': True, 'ignoreHTTPSErrors': True,\
        'defaultViewport': None, 'viewport_width': 800})
```

The above statement creates a "headless" Chromium browser. We intentionally disregard HTTPS errors, but you might prefer to explicitly log the errors instead. We arbitrarily decided to use a viewport of 800 pixels, but allow the page length to be its natural length.

```
context = await browser.createIncognitoBrowserContext()
page = await browser.newPage()
```

The above statements create a new incognito browser page. Next, we *attempt* to disable animation (but this doesn't appear to currently happen the way it should):

```
await page._client.send('Animation.disable')
```

We're now ready to retrieve the page we want to grab, and grab it. Note the "fullPage" setting: we're trying to get the whole thing, not just what might normally be available in an 800x600 pixel viewport:

```
await page.goto(url)
await page.screenshot({'path': myfilespec, 'type': 'jpeg', \
    'quality': 80, 'fullPage': True})
await browser.close()
```

Having grabbed the page, we conclude by setting up the link to the latest version:

```
# ensure the "latest" version is updated for this URL
if (url_or_file == "url"):
    mylatest = mydir + "/" + latestversion
    # print("mylatest =" + mylatest)
elif (url_or_file == "file"):
    mylatest = myfilespec + "_" + "latest" + "." + myextension
try:
    os.unlink(mylatest)
except:
    pass
os.symlink(myfilespec, mylatest)

return(myfilespec)
```

15. DOING OUR DNSDB QUERIES

The next function we're going to look at is the code that handles making queries against DNSDB. This code is an updated version of the code that was discussed in the "Run" part of Farsight's recent "Crawl-Walk-Run" webinar series. We'll discuss it line-by-line. We begin by importing the standard libraries we need:

```
from pathlib import Path
from io import BytesIO
import json
import re
import sys
from time import strftime, gmtime
```

And we're going to use pycurl to make our DNSDB API version 2²¹ calls:

```
import pycurl
```

The `extract_time` function is designed to pull the `time_last` value out of the JSON output we receive. Because we have records both from our sensors AND from ICANN zone files/CZDS files, we need to check for both sorts of `time_last` values since we might have one or the other:

```
# See stackoverflow.com/questions/26924812/python-sort-list-of-json-by-value
def extract_time(myrecord):
    json_format=eval(myrecord)

    try:
        extracted_bit = json_format['obj']['time_last']
    except:
        extracted_bit = json_format['obj']['zone_time_last']

    return extracted_bit
```

The next set of three functions handles formatting our DNSDB output for our three use cases. **The first version handles formatting output for tab #3, our DNSDB RRnames tab.** We begin by extracting the JSON elements we want:

```
def print_bits(myrecord):
    myformat = '%Y-%m-%d %H:%M:%S'
    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)
```

Once again, just as we had to handle both regular `time_last` values and zone file `time_last` values for our sorting function, we also had to handle that for the actual output:

```
try:
    extract_tl = myrecord_json_format['obj']['time_last']
except:
    extract_tl = myrecord_json_format['obj']['zone_time_last']
```

The next section formats that output in our preferred format: `'%Y-%m-%d %H:%M:%S'`

```
tl_datetime = gmtime(extract_tl)
enddatetime = strftime(myformat, tl_datetime)
```

We repeat the extraction process for the `time_first` values:

```
try:
    extract_tf = myrecord_json_format['obj']['time_first']
except:
    extract_tf = myrecord_json_format['obj']['zone_time_first']
```

²¹ <https://docs.dnsdb.info/dnsdb-apiv2/>

```
tf_datetime = gmtime(extract_tf)
startdatetime = strftime(myformat, tf_datetime)
```

Now that we're done getting time values, we extract the counts we want:

```
extract_count = myrecord_json_format['obj']['count']
formatted_count = str('{:>11,d}'.format(extract_count))
```

The preceding format allows for a relatively wide count field, but that's required for some names. We right justify the field, and use comma separators for readability. With all of the preceding out of the way, we then assemble the line, quoting the time values in the output:

```
results = extract_bit + " " + extract_bit_2 + " \"" + enddatetime + \
    "\" \"\" + startdatetime + "\" \" + formatted_count + \
    \" \" + extract_bit_3
```

We're suppressing the SOA records. If you want them, obviously you can just return results unconditionally.

```
if (results.find("SOA") == -1):
    return results
else:
    return ""
```

The second version handles formatting output for our DNSDB Rdata IP query. This routine is very similar to the formatting done for the RRname, but with a wider RRname field:

```
def print_rdata_bits(myrecord):
    myformat = '%Y-%m-%d %H:%M:%S'
    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']
    extract_bit = str('{0:<50}'.format(extract_bit))

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)

    try:
        extract_tl = myrecord_json_format['obj']['time_last']
    except:
        extract_tl = myrecord_json_format['obj']['zone_time_last']

    tl_datetime = gmtime(extract_tl)
    enddatetime = strftime(myformat, tl_datetime)

    try:
        extract_tf = myrecord_json_format['obj']['time_first']
    except:
        extract_tf = myrecord_json_format['obj']['zone_time_first']

    tf_datetime = gmtime(extract_tf)
    startdatetime = strftime(myformat, tf_datetime)

    extract_count = myrecord_json_format['obj']['count']
    formatted_count = str('{:>11,d}'.format(extract_count))
```

```

results = extract_bit + " " + extract_bit_2 + " \'" + enddatetime + \
    "\" \'" + startdatetime + "\" " + formatted_count + \
    " " + extract_bit_3

```

This time we also handle filtering some unwanted "junk" names we noticed. Obviously, this is not a complete list of "junk" names, just enough to show the general process that can be used to suppress names you don't like. We match them using the "re" ("regular expressions") library:

```

unwanted_name_found = False
unwanted_rrnames =
r'(.*\.\verteiltssysteme.net\.$|.*\.\eslared\.org\.\ve\.$|.*\.\usac\.edu\.\gt\.$)'
if re.match(unwanted_rrnames, extract_bit):
    unwanted_name_found = True

if ((results.find("SOA") == -1) and (unwanted_name_found == False)):
    return results
else:
    return ""

```

The third and final version of the formatting routine only extracts limited values from the query output, e.g., the bits we need for our network graph in tab #10:

```

def print_limited_bits(myrecord):

    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    # extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)
    extract_bit_3 = extract_bit_3.replace(' ', '')

    results = extract_bit + " " + extract_bit_2 + " " + extract_bit_3

```

In this third version, we show a different approach to filtering, selecting four RRtypes using anchored regular expressions:

```

rrtypes = r'^ (A|AAAA|CNAME|NS)$'

if re.match(rrtypes, extract_bit_2):
    return results
else:
    return ""

```

The `make_query` routine actually handles doing the DNSDB query. Looking at it line-by-line:

```

def make_query(fqdn, query_type):

```

We begin by retrieving our API key from `~/.\dnsdb-apikey.txt`

```

# get the DNSDB API key
filepath = str(Path.home()) + "/.\dnsdb-apikey.txt"
with open(filepath) as stream:
    myapikey = stream.read().rstrip()

```


Depending on the query type, we'll make one of three queries:

```
if (query_type == "RRname"):
    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rrset/name/" + fqdn
elif (query_type == "RdataIP"):
    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rdata/ip/" + fqdn
elif (query_type == "RdataName"):
    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rdata/name/" + fqdn
```

The requests header will pass the API key and ensure we get JSON Lines-format output returned:

```
requestHeader = []
requestHeader.append('X-API-Key: ' + myapikey)
requestHeader.append('Accept: application/jsonl')
```

The rest of the Pycurl query is pretty typical:

```
buffer = BytesIO()
c = pycurl.Curl()
c.setopt(pycurl.URL, url)
c.setopt(pycurl.HTTPHEADER, requestHeader)
c.setopt(pycurl.WRITEDATA, buffer)
c.perform()
rc = c.getinfo(c.RESPONSE_CODE)
body = buffer.getvalue()
```

The buffer gets returned as bytes; we want a normal Python3 string, so we decode it:

```
content = body.decode('iso-8859-1')
```

Hopefully all will be well (in which case we'll see a 200 return code). If we see something else, that means we won't have results. In that case, just return the integer response code:

```
if rc == 200:
    return content
else:
    return rc
```

We're now ready to handle the entry point for the DNSDB queries. In a nutshell, we begin by getting setup to handle RRname queries OR Rdata queries:

```
def doQuery(fqdn, full_or_limited):
    if ((full_or_limited == "limited") or (full_or_limited == "full")):
        content = make_query(fqdn, "RRname")
    elif (full_or_limited == "RdataIP"):
        content = make_query(fqdn, "RdataIP")
    else:
        print("In dnsbRun.py (doQuery) =" + full_or_limited)
        sys.exit(0)
```

If the DNSDB query goes awry, we'll get a numeric status code instead of DNSDB results. Assuming all went well, we'll break the output into a list, splitting on newline boundaries:

```
try:
    test = int(content)
```

```

    print("Error making dnsdb query! Return code = "+str(test))
    sys.exit(0)
except:
    sList = list(line for line in content.strip().split("\n"))

```

The next chunk of code strips the Farsight Streaming API Framing Protocol²² statements:

```

# we want to dump the first line in that output
# print ("sList[0]="+sList[0])
if sList[0] == '{"cond":"begin"}':
    sList.pop(0)
else:
    print("SOMETHING ODD HAPPENED POPPING THE FIRST ELEMENT")

# print ("sList[-1]="+sList[-1])
if ((sList[-1] == '{"cond":"succeeded"}') or
    (sList[-1] == '{"cond":"limited","msg":"Result limit reached"}')):
    sList.pop()
else:
    print("SOMETHING ODD HAPPENED POPPING THE LAST ELEMENT")

```

Now we're going to sort the remaining results by the time values we extracted:

```
sList2 = sorted(sList, key=extract_time, reverse=True)
```

At this point, we're ready to format output. How we format our output depends a bit on HOW we're going to use it. We have three routines (which you've already seen discussed) to handle this:

```

formatted_output=""
results=""
for line in sList2:
    if (full_or_limited == "full"):
        results=print_bits(line)
    elif (full_or_limited == "limited"):
        results=print_limited_bits(line)
    elif (full_or_limited == "RdataIP"):
        results=print_rdata_bits(line)

    if results != "":
        result_with_nl=results+"\n"
        formatted_output=formatted_output+result_with_nl

if len(formatted_output) == 0:
    formatted_output = "No results found\n"
return(formatted_output)

```

16. HANDLING THE CONTENT FOR THE ASN WHOIS TAB

While the Domain Whois and IP Whois were easily handled via canned third-party libraries, getting ASN Whois (including point-of-contact information) requires a little more effort. There were several issues we had to deal with:

²² <https://docs.dnsdb.info/dnsdb-saf-protocol/>

- Precanned 3rd party routines exist for getting ASN Whois information. If you're looking for something close to what's available for domain names and IP addresses, the closest option may be the ipwhois ASN Whois that leverages Team Cymru ASN information.²³
- Some ASN Whois queries may only return barebones information about the ASN itself, and not anything about the points of contact associated with the ASN. We wanted the "whole thing," as you'd get if you made a command line Whois query for an ASN at the shell prompt.
- Theoretically one could request ASN information from ARIN RWS in multiple formats, including text, JSON and XML.²⁴ Unfortunately, the only ARIN RWS format that returned point of contact information was XML. Okay, XML format it shall be.
- The raw XML output was pretty much unusable as-is, unlike the readily-readable pretty-printed JSON-format Whois output. Converting the raw XML to HTML format required applying the ARIN XML stylesheet to the raw XML. That's not a hard process, IF the XML style sheet specified by the page is valid. Unfortunately, the raw ARIN XML style sheet fails validation as shipped. The fix for this wasn't hard -- once we were able to identify exactly what the issue was.
- We didn't have a lot of experience working with XML libraries, so we basically ended up try most of them before settling on lxml.
- Displaying an HTML file in Tkinter required us to "scrape" the local HTML file.

Bottom line, we got it done, but it felt more painful than it should have been. Let's walk through it:

```
$ cat asnWhois.py
import datetime
import errno
import socket
import os
from pathlib import Path

import dns.resolver
from lxml import etree

import requests
```

With imports out of the way, we now deal with some convenience functions:

```
def confirmDirExists(mydir):
    try:
        os.makedirs(mydir)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise

def confirmFileExists(myfilespec):
    try:
        open(myfilespec, 'a').close()
    except:
        raise

def fqdnToIP(fqdn):
    myip = socket.gethostbyname(fqdn)
```

²³ <https://pypi.org/project/ipwhois/>
<https://ipwhois.readthedocs.io/en/latest/ASN.html>

²⁴ <https://www.arin.net/resources/registry/whois/rws/api/>

```
return(myip)
```

Unlike the last time we needed to map IP addresses to ASNs, this time we're going to use Oregon Routeviews:

```
def reverseIPforRouteviews(myip):
    reversed_ip = ".".join(reversed(myip.split('.')))+"asn.routeviews.org"
    return(reversed_ip)

def getASNfromRouteviews(reversed_ip):
    answers = dns.resolver.resolve(reversed_ip, 'TXT')
    split_answers=answers.response.answer[0].to_text().split(" ")
    myasn=split_answers[4]
    myasn=myasn.replace('\"', '')
    return(myasn)
```

Now we handle making two types of filenames for the ASN Whois work -- raw (XML) and cooked (HTML):

```
def makeOutputFile(fqdn, outputfiletype):
    home = str(Path.home())
    subdir = "asnwhois_output"
    utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'

    if (outputfiletype == "raw"):
        myextension = "xml"
    elif (outputfiletype == "cooked"):
        myextension = "html"

    mydir = home + "/" + subdir
    # if on something non-Un*x-ish, remember os.path.join(dir, f)
    fullfilenamepart = fqdn + "_" + utcdatetime + "." + myextension
    latestversion = fqdn + "_" + "latest" + "." + myextension

    # ensure the directory exists
    confirmDirExists(mydir)

    # ensure the timestamped file exists
    myfilespec = mydir + "/" + fullfilenamepart
    confirmFileExists(myfilespec)

    # set up a convenience link to the latest version
    mylatest = mydir + "/" + latestversion
    makeLink(myfilespec, mylatest)

    return(myfilespec)

def myAsnWhois(fqdn):
    # we need the IP of the FQDN to map to an ASN
    myip = fqdnToIP(fqdn)

    # get the domain we need to get the ASN from Routeviews
    reversed_ip = reverseIPforRouteviews(myip)

    # do IP-->ASN using Routeviews
    myasn = getASNfromRouteviews(reversed_ip)

    # Joint Whois Project allows all queries to go to a single
    # whois server which will redirect as appropriate, see
    # https://www.lacnic.net/1040/2/lacnic/lacnics-whois (we'll use ARIN)
```

```

# now assemble the query URL
myurl = "https://whois.arin.net/rest/asn/" + myasn + "/pft?s=" + myasn

# create the output file to hold the ASN Whois Information
myfilespec = makeOutputFile(fqdn, "raw")

# This time we'll demonstrate using requests instead of pycurl
headers = {'Accept' : 'application/xml'}
response = requests.get(myurl, headers=headers)

# Throw an error for bad status codes
response.raise_for_status()

# if we want the response in unicode, use response.text below
# if we want the response in bytes, use response.content instead

# write the results to a file
with open(myfilespec, "wb") as my_file:
    my_file.write(response.content)

# OK, now we've got an XML copy saved... let's convert it to HTML, save that file and
# return the name of that file...

# IMPORTANT NOTE: we're using a saved copy of the XSLT file because ARIN has
# https://www.w3.org/1999/XSL/Transform BUT THERE SHOULD BE NO "s" there
# (e.g., the URI is regular http not https). If this isn't fixed,
# etree.XSLT will indicated that no stylesheet exists. A subtle bug...
# FWIW, the Oxygen XML Editor immediately found the issue, very impressive!

XSLT_file = str(Path.home()) + "/.bang_question_files/" + "./website.xsl"
transform = etree.XSLT(etree.parse(XSLT_file))
result = transform(etree.parse(myfilespec))
my_transformed_results = etree.tostring(result, pretty_print=True)

cooked_file = makeOutputFile(fqdn, "cooked")

with open(cooked_file, "wb") as outfile:
    outfile.write(my_transformed_results)

return(cooked_file)

```

17. HANDLING DRAWING THE NETWORK GRAPH (THE CONTENT FOR THE FINAL TAB)

Our last substantive content is a network graph for the final tab based on the data from the RRname query. We looked at three graphing packages for this:

- graph-tool²⁵
- igraph²⁶ and
- networkx²⁷

Ultimately, we decided to go with igraph. We'll now explain how we created the content for that tab:

²⁵ <https://git.skewed.de/count0/graph-tool/-/wikis/installation-instructions#homebrew>

²⁶ <https://igraph.org/python/>

²⁷ <https://networkx.org/documentation/stable/index.html>

```
$ cat draw_network_graph.py
```

```
import igraph
import numpy as np
```

The data for the graph is passed in from a specially-limited copy of some DNSDB API output:

```
def draw_the_graph(read_data, myfilespec10):
    record_count = read_data.count('\n')
```

We begin by creating a new directed graph ("directed" in this case meaning the graph edges have a "from --> to" property):

```
# Create a directed graph
g = igraph.Graph(directed=True)
```

Next we're going to build our list of nodes, which igraph refers to as "vertices:"

```
# begin by creating the list of unique vertices
mynodes = []
source_nodes = []
dest_nodes = []
edge_types = []

my_individual_lines = []
my_individual_lines=read_data.split("\n")
for i in range(0,record_count):
    fields=my_individual_lines[i].split(" ")
    if len(fields)==3:
        # remove spaces between rdata elements in fields[2]
        tempfield=fields[2]
        fields[2]=tempfield.replace(" ", "\n")

        # add the node names to the list of vertices
        # a vertex can be either a source or destination node
        mynodes.append(fields[0])
        mynodes.append(fields[2])

        # drop any duplicate vertices
        mynodes = np.ndarray.tolist(np.unique(mynodes))

        # now let's build three lists: sources, destinations and edge_types
        source_nodes.append(fields[0])
        dest_nodes.append(fields[2])
        edge_types.append(fields[1])

mynode_count=len(mynodes)

# create the vertices we need
g.add_vertices(mynode_count)

# populate the vertex properties
for i in range(0,mynode_count):
    g.vs[i]["id"] = i
    g.vs["name"] = mynodes[i]
    g.vs[i]["label"] = mynodes[i]

# now let's create the edges
```

```

for i in range(0, record_count):
    source=mynodes.index(source_nodes[i])
    dest= mynodes.index(dest_nodes[i])
    type= edge_types[i]
    g.add_edges([(source,dest)])
    g.es[i]["label"] = type
    g.es[i]["name"] = type
    g.es[i]["weight"] = 1

visual_style = {}
# Set bbox and margin
visual_style["bbox"] = (800,800)
visual_style["margin"] = 100

# Set vertex colour and size
visual_style["vertex_color"] = 'white'
visual_style["vertex_size"] = 125

# Set vertex label size
visual_style["vertex_label_size"] = 10

# Don't curve the edges
visual_style["edge_curved"] = False

```

Selection of an optimal layout will have a huge impact on the usability of the chart. For most of our test cases, layout_reingold_tilford_circular seems to work quite well.

```

# Set the layout
# my_layout = g.layout_kamada_kawai()
# my_layout = g.layout_circle()
# my_layout = g.layout_drl()
# my_layout = g.layout_fruchterman_reingold()
# lgl = "large graph layout"
# my_layout = g.layout_lgl()
# my_layout = g.layout_random()
# my_layout = g.layout_reingold_tilford()
my_layout = g.layout_reingold_tilford_circular()

visual_style["layout"] = my_layout

# Plot the graph
igraph.plot(g, myfilespec10, **visual_style)

```

18. REMAINING ISSUES

Because this is an experimental/developmental/proof-of-concept application, some bits of this application are incomplete or may have residual issues. This section is meant to capture some of those for posterity.

NOTE: This list of remaining gaps/issues is indicative, but not necessarily comprehensive.

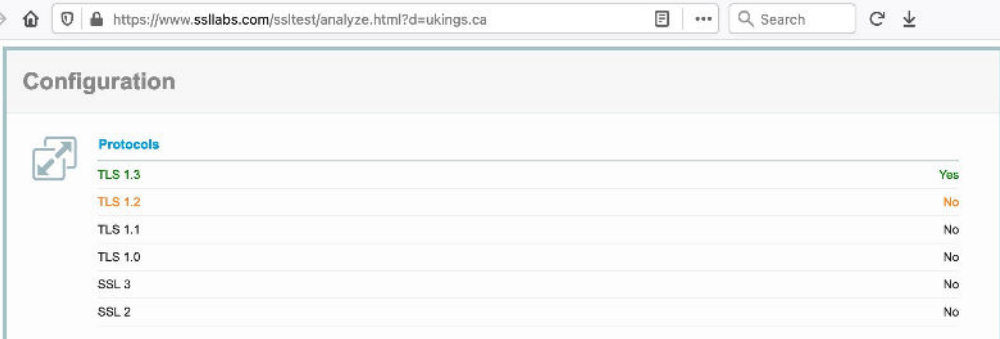
I. General Issues:

- **Application currently uses global variables.** These should be refactored for improve modularity.
- **We do not currently check and cleanly handle all possible error paths.** More comprehensive error handling would improve overall application robustness.

- **Users cannot save or print individual tabs (or entire analyses)**, except to the extent that that content is already being automatically and routinely saved, or user cuts and pastes text content or manually screen captures graphic content.
- Unless systems have effectively-unlimited disk space, **cached screenshots and other elements will eventually need to be manually cleaned up** to avoid disk space exhaustion.
- Users who need the ability to control their apparent endpoint can do so through use of a VPN, but some users might want or need more granular/integrated proxy support. **There is no integrated proxy support currently.**

II. Screen Capture-Related Issues (tab #2):

- **A significant number of sites could not be successfully scraped** due to issues with animations, multimedia, video, or other structural page-related issues (see the sites flagged in the Appendix V). Finding a solution for these currently-unscrapable sites should be a priority.
- Currently the app only handles home pages or other FQDNs accessible on port 443 (https) or port 80 (http). **The application should allow users to specify an arbitrary deep page on a site, including pages served on non-standard arbitrary ports.**
- Sites are not scraped for client-specific environments. That is, **we do offer the ability to masquerade as an iPhone, Android device, etc. when scraping**, even though some sites may offer radically different user experience for different platforms.
- When testing the proof-of-concept application with the site ukings.ca , we found that site only accepts TLS1.3 connections. **Our headless Chromium browser is not TLS1.3-aware, and hence cannot connect to TLS1.3-only sites** (see <https://bugs.chromium.org/p/chromium/issues/detail?id=1054891>). Other sites may also be moving to a TLS1.3-only configuration. See the following test report excerpt (for ukings.ca) from the Qualys SSL Labs site:



Configuration	
Protocols	
TLS 1.3	Yes
TLS 1.2	No
TLS 1.1	No
TLS 1.0	No
SSL 3	No
SSL 2	No

III. DNSDB-Related Issues (tabs #3 and #4):

- **DNSDB passive DNS queries currently run with default parameters and do NOT allow the analyst to specify a specific RRtype, to specify a number of results, to time fence, to specify a bailiwick, etc.**
- We currently have an RRnames tab and an Rdata IP tab, but **no current Rdata Names tab**. We might consider adding an example tab that looks at the name servers for the specified domain, etc.
- **We don't do left hand or right hand wildcard searches of domain delegation points.**
- **We don't do any IP neighborhood searches** (e.g., we currently get the IP whois, but fail to search the encompassing network block)
- **We don't currently do any Flexible Searches.**

- **Sample DNSDB filtering is hardcoded into source as a placeholder**, rather than being user accessible/tailorable.
- Results are always sorted by `time_last_seen` descending. Other sort orders options might be useful (such as by count descending).

IV. Whois-Related Issues (tabs #5, #6 and #7):

- Referrals may not always work. It may be necessary to direct queries specifically to the appropriate regional registry in some cases.
- The Domain Whois and IP Whois output doesn't include point of contact information for resources.
- The pretty-printed JSON of tabs #5 and #6 looks "stark" in comparison to the ARIN ASN Whois tab (#7)
- An XML library that would accept ARIN using https instead of http (as strictly required for the style sheet `http[s]://whois.arin.net/xsl/website.xsl`) would eliminate the need to download and cache a local copy of that file.
- Specifying ASNs in "AS"+integer form will sometimes fail. We currently routinely try the ASN Whois queries without the leading AS prefix as a result.
- It may be worth checking RADB for routing information as well as the regional registries.

V. AS Prefixes (tab #8)

- IPv4 and IPv6 prefixes are comingled post-sort. It would be preferable to have separately sorted IPv4 and IPv6 prefixes.
- It would be desirable to indent more specific routes when covering routes are also announced.
- Being able to click on prefixes to pivot to that prefix would likely be handy.

VI. Geolocation Map (tab #9)

- Two projections are currently enabled, allowing a US view or a world view, but we suspect that Europeans would likely appreciate a more geographically-appropriate projection, ditto Asians, South Americans, Africans, Australians/New Zealanders, etc.
- Selection of one projection or another is *ad hoc* at best, based on approximate lat/lon map coverage.
- Label the location with the domain name, IP or city name (currently there's just a dot)
- The outline map is currently rather stark. Maybe use a satellite picture instead?
- Geolocation looks "wrong" (even though it's right) if a CDN or cloud provider is in use (e.g., an East Coast US School may show up in California or Seattle due to use of a CDN or cloud provider). Explicitly annotate that CDN or cloud provider status as part of the map marginalia?

VII. Network Graph (tab #10)

- Labeling the graph is a challenge since labels may be short and sweet (perhaps just a brief IP address such as 8.8.8.8) or painfully long (such as a multi-record IPv6 record entry).
- The current graph is really just a proof-of-concept and isn't very exciting since the graph always has a common origin e.g., the queried FQDN (in some stable cases, the graph may just be a two or three node graph). Augmenting the basic RRname query data that's currently feeding the graph with data from "canned pivots" would likely make the graphs more visually appealing.
- Being able to drag-and-drop or manipulate nodes would likely be well received, ditto the ability to "dig in" on potentially interesting nodes.
- When testing with **www.stanford.edu**, the network graph is effectively unusable due to the large number of unique RRsets associated with that RRname. The result ends up looking like a bullseye...



VIII. Totally New Functionality (Potential Additional Tabs)

Note: currently the window just gets wider and wider as additional tabs are added. It would be ideal if a row of tabs could be "scrolled", or tabs could be stacked. If that's not possible, it might be necessary to have a hierarchical "notebook of notebooks." Notwithstanding that constraint, some potential additional tabs might include:

- Collect and report **affiliate tags and cookies**.
- **Alexa rank, age and value** for the domain
- Users currently submit a single site at a time, which then runs synchronously through each step. Imagine an alternative "**batch mode**" where the user could submit a file full of URLs for processing, and then browse through a stack of completed runs at their leisure.
- **Characterize the domain or IP**. Is it a:
 - CDN?
 - End User Dynamic IP address?
 - End User Dynamic domain name?
 - Free email domain?
 - Known sinkhole IP?
 - Parked domain?
 - Tor endpoint?
- **Domain and/or IP reputation reporting** (is a given domain or IP on any blocklists, allowlists, etc.)? Evidence of any problematic content? Spam? Malware hits? Phishing hits?
- Collect and check **hashes for all graphics** in an effort to find commonalities with other pages or to identify known problematic content.
- **Identify language of home page; automatically translate to a preferred language if not currently in that language.**
- **Nmap** results for domain's IP (note: active reconnaissance may trigger technical or legal response)
- Tasks that create content for the various tabs currently run serially. Imagine **parallelizing those tasks**. This should reduce the overall time-to-complete a given run.
- **Persistent "Lab notebook" page** allowing the user to document their analyses and observations.
- **SSL/TLS certificate retrieval and potential pivoting.**
- **Traffic volume over time**
- **Wayback Machine:** does the site look the same as it used to? Imagine a page that would link to appropriate archive.org pages, including some statistical measure of visual similarity.

APPENDICIES

APPENDIX I. INSTALLATION

(1) Unpack the bang_question.tar.gz compressed tar file you've downloaded:

```
$ tar xfvz bang_question.tar.gz
```

(2) cd into the bang_question directory created in the previous step:

```
$ cd bang_question
```

(3) Install required prerequisite Python3 libraries with pip3:

```
$ /usr/local/bin/pip3 install --user .
```

Note the trailing space and dot on that command!

(4) Install Chromium:

```
$ pypeteer-install
```

(5) Make a directory called ~/.bang_question_files

```
$ mkdir ~/.bang_question_files
```

Note the squiggle slash dot before bang!

(6) Copy exclamationquestion.gif to the ~/.bang_question_files/ directory:

```
$ cp exclamationquestion.gif ~/.bang_question_files/
```

(7) Using wget (or a web browser of your choice) download and save a copy of <https://whois.arin.net/xsl/website.xsl>

```
$ wget "https://whois.arin.net/xsl/website.xsl"
```

Copy that file to the ~/.bang_question_files/ directory:

```
$ cp website.xsl ~/.bang_questions/
```

(8) (a) Download a recent Routing Information Base file from Oregon Routeviews:

```
$ pyasn_util_download.py --latest
```

That will download a file such as rib.20210104.2200.bz2

(b) Convert that file to the "ipasn_db_file" file we actually need with:

```
$ pyasn_util_convert.py --single rib.20210104.2200.bz2 my_ip2asn_db_file
```

Obviously, substitute the name of the actual RIB file you downloaded for the value shown here.

(c) Move `my_ip2asn_db_file` to `~/ .bang_question_files/`

```
$ mv my_ip2asn_db_file ~/ .bang_question_files/
```

(9) (a) Sign up for a free IP2location account at <https://lite.ip2location.com/>

(b) Download and unzip a copy of DB9LITEBINIPV6 (note that this is the **binary** IPv4 **AND** IPv6 format database, notwithstanding the name!) from <https://lite.ip2location.com/>

(c) Move the downloaded and unzipped file `IP2LOCATION-LITE-DB9.IPV6.BIN` to the `~/ .bang_question_files`

```
$ mv IP2LOCATION-LITE-DB9.IPV6.BIN ~/ .bang_question_files
```

NOTE: this file may unpack into `IP2LOCATION-LITE-DB9.IPV6.BIN/IP2LOCATION-LITE-DB9.IPV6.BIN` -- you only want to move the FILE, NOT the file AND the directory by the same name, to the new target location!

(10) While in the subdirectory with the `bang_question.py` file, create an alias pointing at `bang_question.py`:

```
$ alias bang_question=`pwd`/bang_question.py
```

You can then simply say:

```
$ bang_question
```

to run the application. If you just type:

```
$ alias
```

you'll see the alias you've defined for `bang_question`. For example:

```
$ alias
alias bang_question='/Users/joe/bang_question/bang_question.py'
```

You may want to add the value you see for your installation to your `~/ .bash_profile` (or similar startup file) to make your alias persistent.

(11) Access to DNSDB API is controlled by an API key. Our code looks for your key in `~/ .dnsdb-apikey.txt` (note the leading dot on that file name!)

When creating that file, just put the key in the file (no quotes or other prefatory content is required).

If you need to arrange to get a DNSDB API key, see:

- <https://www.farsightsecurity.com/dnsdb-community-edition/>
- <https://www.farsightsecurity.com/trial-api/>
- <https://www.farsightsecurity.com/order-form/>

APPENDIX II. SOURCE CODE FOR THE FULL PROOF-OF-CONCEPT APPLICATION

```
$ cat setup.py
from setuptools import setup

setup(
    name="bang_question",
    version="0.1",
    description="Proof-of-concept cyber investigative framework",
    author="Joe St Sauver",
    author_email="stsauver@fsi.io",
    scripts=['bin/bang_question'],
    install_requires=[
        'dnspython',
        'ipwhois',
        'ip2location',
        'kaleido',
        'lxml',
        'numpy',
        'pandas',
        'Pillow',
        'plotly-express',
        'pyasn',
        'pycurl',
        'PyMuPDF',
        'pyppeteer2',
        'python-igraph',
        'requests',
        'whois',
    ],
    license="Apache Software License",
    packages=setuptools.find_packages(),
    classifiers=[
        "License :: OSI Approved :: Apache Software License",
        "Programming Language :: Python :: 3.9",
    ],
    python_requires='>=3.9.1',
)
```

```
$ cat bang_question.py
#!/usr/local/bin/python3
# you MUST install Python 3.9.1 (or later) on macOS Big Sur
# (see https://www.python.org/downloads/release/python-391/ )

import asyncio
import errno
import json
import os
from pathlib import Path
import socket
import sys

from datetime import datetime
import datetime

import tkinter as tk
from tkinter import NW, TOP, LEFT, END
```

```

from tkinter.messagebox import *

from tkinter.ttk import *
import tkinter.ttk as ttk

import PIL
from PIL import Image, ImageTk

# https://pypi.org/project/pyppeteer2/
# https://miyakogi.github.io/pyppeteer/reference.html
# One time, download Chromium: $ pyppeteer-install
# from pyppeteer import *

### This site or product includes IP2Location LITE data available
### from https://lite.ip2location.com
### We assume that a local copy of IP2LOCATION-LITE-DB9.IPV6.BIN
### has been downloaded and installed. (That file is for both V4 & V6)
import IP2Location

import plotly.express as px
import kaleido

import pandas as pd

import pyasn

from ipwhois import IPWhois

# https://github.com/DannyCork/python-whois
import whois

# https://pypi.org/project/PyMuPDF/
# https://pymupdf.readthedocs.io/en/latest/
import fitz

# make sure you don't include the .py as part of the name of the imports
from scrapePage import scrapeAFQDN
from asnWhois import myAsnWhois
from dnsdbRun import doQuery
from draw_network_graph import draw_the_graph

##### Globals #####

global root, mywindow, mynotebook, mymenubarbox, mymenubarlabel, my_log_box
global fqdn, FQDN, myip, buttona, buttonb, buttonc, mymapname9
global t1, t2, t3, t4, t5, t6, t7, t8, t9, t10
global background2, img4, t10img

##### Functions #####

# https://stackoverflow.com/questions/56043767/show-large-image-using-scrollbar-in-python/56043976
class ScrollableImage(tk.Frame):
    def __init__(self, master=None, **kw):
        self.image = kw.pop('image', None)
        sw = kw.pop('scrollbarwidth', 10)
        super(ScrollableImage, self).__init__(master=master, **kw)
        self.cnvs = tk.Canvas(self, highlightthickness=0, **kw)
        self.cnvs.create_image(0, 0, anchor='nw', image=self.image)

```



```

# Vertical and Horizontal scrollbars
self.v_scroll = tk.Scrollbar(self, orient='vertical', width=sw)
self.h_scroll = tk.Scrollbar(self, orient='horizontal', width=sw)
# Grid and configure weight.
self.cnvs.grid(row=0, column=0, sticky='nsew')
self.h_scroll.grid(row=1, column=0, sticky='ew')
self.v_scroll.grid(row=0, column=1, sticky='ns')
self.rowconfigure(0, weight=1)
self.columnconfigure(0, weight=1)
# Set the scrollbars to the canvas
self.cnvs.config(xscrollcommand=self.h_scroll.set,
                 yscrollcommand=self.v_scroll.set)
# Set canvas view to the scrollbars
self.v_scroll.config(command=self.cnvs.yview)
self.h_scroll.config(command=self.cnvs.xview)
# Assign the region to be scrolled
self.cnvs.config(scrollregion=self.cnvs.bbox('all'))
self.cnvs.bind_class(self.cnvs, "<MouseWheel>", self.mouse_scroll)

def mouse_scroll(self, evt):
    if evt.state == 0 :
        self.cnvs.yview_scroll(-1*(evt.delta), 'units') # For MacOS
        self.cnvs.yview_scroll(int(-1*(evt.delta/120)), 'units') # For windows
    if evt.state == 1:
        self.cnvs.xview_scroll(-1*(evt.delta), 'units') # For MacOS
        self.cnvs.xview_scroll(int(-1*(evt.delta/120)), 'units') # For windows

### Callback for submit button
def funca(event=None):
    global fqdn, mymapname9, myip, my_log_box
    global background2, img, img2, img4, t10img
    global t1, t2, t3, t4, t5, t6, t7, t8, t9, t10

    # we log events as we have them to report...
    # clear any existing log entries
    my_log_box.delete('1.0', END)

    my_log_box.insert(tk.END, "New run initiated\n")
    my_log_box.update()

    # callback for Submit button -- FQDN field should be filled in now
    fqdn = FQDN.get()

    my_log_box.insert(tk.END, "The FQDN entered was "+fqdn+"\n")
    my_log_box.update()

    # -----

    my_log_box.insert(tk.END, "About to do screen grab of FQDN...\n")
    my_log_box.update()

    # check the supplied domain, and try to grab a copy of the specified page
    mypage = asyncio.get_event_loop().run_until_complete\
        (scrapeAFQDN(fqdn, my_log_box, "url"))

    # there was a problem scraping the page
    if (str(mypage) == "1"):
        return(1)

```

```

my_log_box.insert(tk.END, " Screen grab done. Grabbed page is\n "+\
    str(mypage)+"\n")
my_log_box.update()

try:
    t2.destroy()
except:
    pass

t2 = tk.Frame()
mynotebook.add(t2, text="Screen Grab")
img2 = ImageTk.PhotoImage(Image.open(mypage))
image_window = ScrollableImage(t2, image=img2,
    width=800, height=600)
image_window.pack(anchor=NW)
image_window.update()

my_log_box.insert(tk.END, " Screen grab added to tab.\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Running DNSDB RRname query now...\n")
my_log_box.update()

my_fqdn_results_json = doQuery(fqdn, "full")

try:
    t3.destroy()
except:
    pass

t3 = tk.Frame()
mynotebook.add(t3, text="DNSDB RRnames")

# Text Widget height and width are in characters
mytext_widget_pdns = tk.Text(t3,height=40,width=132)
mytext_widget_pdns.pack(side=TOP, anchor=NW)
mytext_widget_pdns.update()

mytext_widget_pdns.insert(tk.END, my_fqdn_results_json)
my_log_box.insert(tk.END, " Passive DNS results added to tab.\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Running DNSDB Rdata IP query now...\n")
my_log_box.update()

myip = socket.gethostbyname(fqdn)
my_ip_results_json = doQuery(myip, "RdataIP")

try:
    t4.destroy()
except:
    pass

t4 = tk.Frame()
mynotebook.add(t4, text="DNSDB IP Rdata")

```

```

# Text Widget height and width are in characters
mytext_widget_pdns_2 = tk.Text(t4,height=40,width=132)
mytext_widget_pdns_2.pack(side=TOP, anchor=NW)
mytext_widget_pdns_2.update()

mytext_widget_pdns_2.insert(tk.END, my_ip_results_json)
my_log_box.insert(tk.END, " Passive DNS Rdata IP results added to tab.\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Now working on Domain WHOIS...\n")
my_log_box.update()

domain = whois.query(fqdn)
domain_json = json.dumps(domain.__dict__, indent=4, default=str)

try:
    t5.destroy()
except:
    pass

t5 = tk.Frame()
mynotebook.add(t5, text="DomWhois")
mytext_widget_5 = tk.Text(t5,height=40,width=132)
mytext_widget_5.pack(side=TOP, anchor=NW)
mytext_widget_5.insert(tk.END, domain_json)

my_log_box.insert(tk.END, " Domain WHOIS results added to tab.\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Now working on IP WHOIS...\n")
my_log_box.update()

myip = socket.gethostbyname(fqdn)
obj = IPWhois(myip)
res=obj.lookup_whois(get_referral=True)
pretty_printed_text = json.dumps(res, indent=4)

try:
    t6.destroy()
except:
    pass

t6 = tk.Frame()
mynotebook.add(t6, text="IPWhois")
mytext_widget_6 = tk.Text(t6,height=40,width=132)
mytext_widget_6.pack(side=TOP, anchor=NW)
mytext_widget_6.insert(tk.END, pretty_printed_text)

my_log_box.insert(tk.END, " IP WHOIS results added to tab.\n")
my_log_box.update()

# -----

# get the basic ASN whois info

```

```

my_log_box.insert(tk.END, "Now working on ASN WHOIS results...\n")
my_log_box.update()

# scraping the ASN Whois
my_asn_info_file = myAsnWhois(fqdn)
myasnpage = asyncio.get_event_loop().run_until_complete\
    (scrapeAFQDN(my_asn_info_file, my_log_box, "file"))

try:
    t7.destroy()
except:
    pass

t7 = tk.Frame()
mynotebook.add(t7, text="ASNWhois")
asnim7 = ImageTk.PhotoImage(Image.open(myasnpage))
image_window = ScrollableImage(t7, image=asnim7,
    width=1000, height=600)
image_window.pack(anchor=NW)
image_window.update()

my_log_box.insert(tk.END, " ASN Whois info added to tab."+"\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Now working on Adding prefixes...\n")
my_log_box.update()

# we need the ASN for some of the filenames, so we'll get that first
myip2asnfilespec = str(Path.home()) + "/.bang_question_files/" + \
    "my_ip2asn_db_file"
asndb = pyasn.pyasn(myip2asnfilespec)
asn = asndb.lookup(myip)

# get the prefixes associated with the ASN
prefixes = asndb.get_as_prefixes(asn[0])
prefixes = sorted(prefixes)

combined_text=""
for pre in prefixes:
    combined_text=combined_text+"\n"+pre

try:
    t8.destroy()
except:
    pass

t8 = tk.Frame()
mynotebook.add(t8, text="Prefixes")
mytext_widget_8_prefixes = tk.Text(t8,height=40,width=132)
mytext_widget_8_prefixes.pack(side=TOP, anchor=NW)
mytext_widget_8_prefixes.insert(tk.END, "AS"+str(asn[0])+" Prefixes:\n")
mytext_widget_8_prefixes.insert(tk.END, combined_text)

my_log_box.insert(tk.END, " ASN Prefixes added to tab.\n")
my_log_box.update()

# -----

```

```

my_log_box.insert(tk.END, "Now working on geolocation..\n")
my_log_box.update()

home = str(Path.home())
subdir9 = "geolocation-maps"
myfilename9 = fqdn
utcdatetime9 = datetime.datetime.utcnow().isoformat() + 'Z'
myextension9 = "jpeg"
mymapextension9 = "png"
mydir9 = home + "/" + subdir9
fullfilenamepart9 = fqdn + "_" + utcdatetime9 + "." + myextension9
latestversion9 = fqdn + "_" + "latest" + "." + myextension9
mymapname9 = mydir9 + "/" + fqdn + "_" + utcdatetime9 + "." + \
    mymapextension9
myfilespec9 = mydir9 + "/" + fullfilenamepart9

# ensure the directory exists
try:
    os.makedirs(mydir9)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

try:
    open(myfilespec9, 'a').close()
except:
    raise

try:
    open(mymapname9, 'a').close()
except:
    raise

geoipfilespec = str(Path.home()) + "/.bang_question_files/" + \
    "IP2LOCATION-LITE-DB9.IPV6.BIN"
database = IP2Location.IP2Location(geoipfilespec, "SHARED_MEMORY")
rec = database.get_all(myip)
lat = rec.latitude
lon = rec.longitude
cit = rec.city

# map it (we need to make it into a DataFrame first
d={"txt":cit,"lat":lat,"lon":lon,"siz":10}
df=pd.DataFrame(d,columns=["txt","lat","lon","siz"])

if ((lon >= -180) and (lon <= -52) and
    (lat >= 36) and (lat <= 83)):
    fig=px.scatter_geo(df,lat="lat",lon="lon",text="txt",\
        size="siz",projection="albers usa",width=800,height=600)
else:
    fig=px.scatter_geo(df,lat="lat",lon="lon",text="txt",\
        size="siz",projection="mercator",width=800,height=600)

fig.write_image(mymapname9)

# ensure the image is appropriately sized
map_img=Image.open(mymapname9).resize((800, 600),Image.ANTIALIAS)
map_img.load()

```

```

# BUG: https://stackoverflow.com/questions/42099914/imagetk-photoimage-doesnt-show-
up-on-osx-but-does-on-windows
# See https://stackoverflow.com/questions/41576637/are-rgba-pngs-unsupported-in-
python-3-5-pillow
# See Yuji Tomita's post at https://stackoverflow.com/questions/9166400/convert-rgba-
png-to-rgb-with-pil
background2 = Image.new("RGB", map_img.size, (255,255,255))
background2.paste(map_img, mask=map_img.split()[3]) # 3 is the alpha channel
background2.save(myfilespec9, 'JPEG', quality=80)
img4 = ImageTk.PhotoImage(background2)

try:
    t9.destroy()
except:
    pass

t9 = tk.Frame()
mynotebook.add(t9, text="GeoIP")
panel2 = tk.Label(t9, image=img4)
panel2.pack(side=TOP, anchor=NW)
mynotebook.add(t9)

my_log_box.insert(tk.END, " Geolocation map added to tab.\n")
my_log_box.update()

# -----

my_log_box.insert(tk.END, "Working on network graph...\n")
my_log_box.update()

subdir10 = "network-graphs"
myfilename10 = fqdn
utcdatetime10 = datetime.datetime.utcnow().isoformat() + 'Z'
myextension10 = "pdf"
mydir10 = home + "/" + subdir9
fullfilenamepart10 = fqdn + "_" + utcdatetime10 + "." + myextension10
myfilespec10 = mydir10 + "/" + fullfilenamepart10
latestversion10 = fqdn + "_" + "latest" + "." + myextension10

# ensure the directory exists
try:
    os.makedirs(mydir10)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

try:
    open(myfilespec10, 'a').close()
except:
    raise

draw_the_graph(doQuery(fqdn, "limited"), myfilespec10)
doc=fitz.open(myfilespec10)
page=doc.loadPage(0)
pix=page.getPixmap()
mode="RGB"

try:
    t10.destroy()

```

```

except:
    pass

t10 = tk.Frame()
mynotebook.add(t10, text="Graph")
t10img=Image.frombytes(mode,[pix.width, pix.height], pix.samples)
t10tkimg=ImageTk.PhotoImage(t10img)
t10_image_window = ScrollableImage(t10, image=t10tkimg,
    width=800, height=600)
t10_image_window.pack(anchor=NW)

my_log_box.insert(tk.END, " Network graph added to tab.\n")
my_log_box.update()

my_log_box.insert(tk.END, "--DONE--\n")
my_log_box.update()

def cleanup_tab(tab_to_cleanup):
    try:
        tab_to_cleanup.destroy()
    except:
        pass

def funcb(event=None):
    global FQDN, my_log_box
    global t1, t2, t3, t4, t5, t6, t7, t8, t9, t10

    FQDN.delete(0, 'end')

    my_log_box.delete('1.0', END)
    my_log_box.update()

    # we intentionally clean these up in reverse order and leave t1 alone
    my_tabs = [t10, t9, t8, t7, t6, t5, t4, t3, t2]
    for i in my_tabs:
        cleanup_tab(i)

### Callback for quit button
def funcc(event=None):
    sys.exit()

##### MAIN #####

def main():
    root = tk.Tk()
    root.title("bang_question")
    iconfilespec = str(Path.home()) + "/.bang_question_files/" + \
        "exclamationquestion.gif"
    root.tk.call('wm', 'iconphoto', root._w, ImageTk.PhotoImage(file=iconfilespec))
    s = ttk.Style()
    s.configure('TNotebook', tabposition='nw')
    s.theme_use('clam')
    root.bind('<Return>', funca)
    mywindow = tk.Frame(root)

    # create notebook
    global mynotebook
    mynotebook = ttk.Notebook(mywindow)
    mynotebook.pack(side=TOP, anchor=NW)

```

```

t1 = tk.Frame()
mynotebook.add(t1, text="MAIN")

# build menu box
mymenubarbox = ttk.Frame(t1)
mymenubarbox.pack(side=TOP, anchor=NW)
mymenubarlabel = ttk.Label(mymenubarbox, text="    Enter FQDN:")
mymenubarlabel.pack(side=LEFT)
global FQDN
FQDN = ttk.Entry(mymenubarbox)
FQDN.pack(side=LEFT)
buttona = tk.Button(mymenubarbox, text='Submit', command = funca)
buttona.pack(side=LEFT)
buttonb = tk.Button(mymenubarbox, text='Clear', command = funcb)
buttonb.pack(side=LEFT)
buttonc = tk.Button(mymenubarbox, text='Quit', command = funcc)
buttonc.pack(side=LEFT)
mymenubarbox.pack(side=TOP, anchor=NW)

# build log box
global my_log_box
my_log_box = tk.Text(t1,height=40,width=132)
my_log_box.pack(side=TOP, anchor=NW)

# add tab #1 to notebook
mynotebook.add(t1, text="MAIN")
mynotebook.pack(side=TOP, anchor=NW, expand=True)

mywindow.pack(side=TOP, anchor=NW)
mywindow.mainloop()

if __name__ == "__main__":
    # execute only if run as a script
    main()

```

```
$ cat asnWhois.py
```

```

import datetime
import errno
import socket
import os
from pathlib import Path
import dns.resolver
from lxml import etree
import requests

def confirmDirExists(mydir):
    try:
        os.makedirs(mydir)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise

def confirmFileExists(myfilespec):
    try:
        open(myfilespec, 'a').close()
    except:
        raise

```



```

def makeLink(myfilespec, mylatest):
    try:
        os.unlink(mylatest)
    except:
        pass
    os.symlink(myfilespec, mylatest)

def fqdnToIP(fqdn):
    myip = socket.gethostbyname(fqdn)
    return(myip)

def reverseIPforRouteviews(myip):
    reversed_ip = ".".join(reversed(myip.split('.')))+".asn.routeviews.org"
    return(reversed_ip)

def getASNfromRouteviews(reversed_ip):
    answers = dns.resolver.resolve(reversed_ip, 'TXT')
    split_answers=answers.response.answer[0].to_text().split(" ")
    myasn=split_answers[4]
    myasn=myasn.replace('\"', '')
    return(myasn)

def makeOutputFile(fqdn, outputfiletype):
    home = str(Path.home())
    subdir = "asnwhois_output"
    utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'

    if (outputfiletype == "raw"):
        myextension = "xml"
    elif (outputfiletype == "cooked"):
        myextension = "html"

    mydir = home + "/" + subdir
    # if on something non-Un*x-ish, remember os.path.join(dir, f)
    fullfilenamepart = fqdn + "_" + utcdatetime + "." + myextension
    latestversion = fqdn + "_" + "latest" + "." + myextension

    # esure the directory exists
    confirmDirExists(mydir)

    # ensure the timestamped file exists
    myfilespec = mydir + "/" + fullfilenamepart
    confirmFileExists(myfilespec)

    # set up a convenience link to the latest version
    mylatest = mydir + "/" + latestversion
    makeLink(myfilespec, mylatest)

    return(myfilespec)

def myAsnWhois(fqdn):
    # we need the IP of the FQDN to map to an ASN
    myip = fqdnToIP(fqdn)

    # get the domain we need to get the ASN from Routeviews
    reversed_ip = reverseIPforRouteviews(myip)

    # do IP-->ASN using Routeviews
    myasn = getASNfromRouteviews(reversed_ip)

```

```

# Joint Whois Project allows all queries to go to a single
# whois server which will redirect as appropriate, see
# https://www.lacnic.net/1040/2/lacnic/lacnics-whois (we'll use ARIN)

# now assemble the query URL
myurl = "https://whois.arin.net/rest/asn/" + myasn + "/pft?s=" + myasn

# create the output file to hold the ASN Whois Information
myfilespec = makeOutputFile(fqdn, "raw")

# If using requests instead of pycurl
headers = {'Accept' : 'application/xml'}
response = requests.get(myurl, headers=headers)

# Throw an error for bad status codes
response.raise_for_status()

# if we want the response in unicode, use response.text below
# if we want the response in bytes, use response.content instead

# write the results to a file
with open(myfilespec, "wb") as my_file:
    my_file.write(response.content)

# IMPORTANT NOTE: we're using a saved copy of XSLT because ARIN shows
# https://www.w3.org/1999/XSL/Transform BUT THERE SHOULD BE NO "s" there
# (e.g., the URI MUST BE regular http not https). If this isn't fixed,
# etree.XSLT will indicated that no stylesheet exists. A subtle bug...
# FWIW, the Oxygen XML Editor immediately found the issue, very impressive!

XSLT_file = str(Path.home()) + "/.bang_question_files/" + "./website.xsl"
transform = etree.XSLT(etree.parse(XSLT_file))
result = transform(etree.parse(myfilespec))
my_transformed_results = etree.tostring(result, pretty_print=True)

cooked_file = makeOutputFile(fqdn, "cooked")

with open(cooked_file, "wb") as outfile:
    outfile.write(my_transformed_results)

return(cooked_file)

```

```
$ cat dnsdbRun.py
```

```

from pathlib import Path
from io import BytesIO
import json
import re
import sys
from time import strftime, gmtime

import pycurl

# See stackoverflow.com/questions/26924812/python-sort-list-of-json-by-value
def extract_time(myrecord):
    json_format=eval(myrecord)

    try:
        extracted_bit = json_format['obj']['time_last']

```

```

except:
    extracted_bit = json_format['obj']['zone_time_last']

return extracted_bit

def print_bits(myrecord):
    myformat = '%Y-%m-%d %H:%M:%S'
    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)

    try:
        extract_tl = myrecord_json_format['obj']['time_last']
    except:
        extract_tl = myrecord_json_format['obj']['zone_time_last']

    tl_datetime = gmtime(extract_tl)
    enddatetime = strftime(myformat, tl_datetime)

    try:
        extract_tf = myrecord_json_format['obj']['time_first']
    except:
        extract_tf = myrecord_json_format['obj']['zone_time_first']

    tf_datetime = gmtime(extract_tf)
    startdatetime = strftime(myformat, tf_datetime)

    extract_count = myrecord_json_format['obj']['count']
    formatted_count = str('{:>11,d}'.format(extract_count))
    results = extract_bit + " " + extract_bit_2 + " \'" + enddatetime + \
        "\' \'" + startdatetime + "\' " + formatted_count + \
        " " + extract_bit_3

    if (results.find("SOA") == -1):
        return results
    else:
        return ""

def print_rdata_bits(myrecord):
    myformat = '%Y-%m-%d %H:%M:%S'
    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']
    extract_bit = str('{0:<50}'.format(extract_bit))

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)

    try:
        extract_tl = myrecord_json_format['obj']['time_last']
    except:
        extract_tl = myrecord_json_format['obj']['zone_time_last']

```

```

tl_datetime = gmtime(extract_tl)
enddatetime = strftime(myformat, tl_datetime)

try:
    extract_tf = myrecord_json_format['obj']['time_first']
except:
    extract_tf = myrecord_json_format['obj']['zone_time_first']

tf_datetime = gmtime(extract_tf)
startdatetime = strftime(myformat, tf_datetime)

extract_count = myrecord_json_format['obj']['count']
formatted_count = str('{:>11,d}'.format(extract_count))
results = extract_bit + " " + extract_bit_2 + " \'" + enddatetime + \
    "\" \'" + startdatetime + "\" " + formatted_count + \
    " " + extract_bit_3

unwanted_name_found = False
unwanted_rrnames =
r'(.*\.\verteiltssysteme.net\.$|.*\.\eslared\.org\.\ve\.$|.*\.\usac\.edu\.gt\.$)'
if re.match(unwanted_rrnames, extract_bit):
    unwanted_name_found = True

if ((results.find("SOA") == -1) and (unwanted_name_found == False)):
    return results
else:
    return ""

def print_limited_bits(myrecord):

    myrecord_json_format = json.loads(myrecord)
    extract_bit = myrecord_json_format['obj']['rrname']

    extract_bit_2 = myrecord_json_format['obj']['rrtype']
    # extract_bit_2 = str('{0:<5}'.format(extract_bit_2))

    temp_bit_3 = myrecord_json_format['obj']['rdata']
    extract_bit_3 = json.dumps(temp_bit_3)
    extract_bit_3 = extract_bit_3.replace(' ', '')

    results = extract_bit + " " + extract_bit_2 + " " + extract_bit_3

    rrtypes = r'^ (A|AAAA|CNAME|NS)$'

    if re.match(rrtypes, extract_bit_2):
        return results
    else:
        return ""

def make_query(fqdn, query_type):
    # get the DNSDB API key
    filepath = str(Path.home()) + "/.dnsdb-apikey.txt"
    with open(filepath) as stream:
        myapikey = stream.read().rstrip()

    if (query_type == "RRname"):
        url = "https://api.dnsdb.info/dnsdb/v2/lookup/rrset/name/" + fqdn
    elif (query_type == "RdataIP"):

```

```

    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rdata/ip/" + fqdn
elif (query_type == "RdataName"):
    url = "https://api.dnsdb.info/dnsdb/v2/lookup/rdata/name/" + fqdn

requestHeader = []
requestHeader.append('X-API-Key: ' + myapikey)
requestHeader.append('Accept: application/jsonl')

buffer = BytesIO()
c = pycurl.Curl()
c.setopt(pycurl.URL, url)
c.setopt(pycurl.HTTPHEADER, requestHeader)
c.setopt(pycurl.WRITEDATA, buffer)
c.perform()
rc = c.getinfo(c.RESPONSE_CODE)
body = buffer.getvalue()
content = body.decode('iso-8859-1')

if rc == 200:
    return content
else:
    return rc

def doQuery(fqdn, full_or_limited):

    if ((full_or_limited == "limited") or (full_or_limited == "full")):
        content = make_query(fqdn, "RRname")
    elif (full_or_limited == "RdataIP"):
        content = make_query(fqdn, "RdataIP")
    else:
        print("In dnsbRun.py (doQuery) =" + full_or_limited)
        sys.exit(0)

    try:
        test = int(content)
        print("Error making dnsdb query! Return code = " + str(test))
        sys.exit(0)
    except:
        sList = list(line for line in content.strip().split("\n"))

        # we want to dump the first line in that output
        # print ("sList[0]=" + sList[0])
        if sList[0] == '{"cond": "begin"}':
            sList.pop(0)
        else:
            print("SOMETHING ODD HAPPENED POPPING THE FIRST ELEMENT")

        # print ("sList[-1]=" + sList[-1])
        if ((sList[-1] == '{"cond": "succeeded"}') or
            (sList[-1] == '{"cond": "limited", "msg": "Result limit reached"}')):
            sList.pop()
        else:
            print("SOMETHING ODD HAPPENED POPPING THE LAST ELEMENT")

        sList2 = sorted(sList, key=extract_time, reverse=True)

        formatted_output=""
        results=""
        for line in sList2:

```

```

    if (full_or_limited == "full"):
        results=print_bits(line)
    elif (full_or_limited == "limited"):
        results=print_limited_bits(line)
    elif (full_or_limited == "RdataIP"):
        results=print_rdata_bits(line)

    if results != "":
        result_with_nl=results+"\n"
        formatted_output=formatted_output+result_with_nl

    if len(formatted_output) == 0:
        formatted_output = "No results found\n"
    return(formatted_output)

```

\$ cat draw_network_graph.py

```

import igraph
import numpy as np

def draw_the_graph(read_data, myfilespec10):
    record_count = read_data.count('\n')

    # Create a directed graph
    g = igraph.Graph(directed=True)

    # begin by creating the list of unique vertices
    mynodes = []
    source_nodes = []
    dest_nodes = []
    edge_types = []

    my_individual_lines = []
    my_individual_lines=read_data.split("\n")
    for i in range(0,record_count):
        fields=my_individual_lines[i].split(" ")
        if len(fields)==3:
            # remove spaces between rdata elements in fields[2]
            tempfield=fields[2]
            fields[2]=tempfield.replace(" ", "\n")

            # add the node names to the list of vertices
            # a vertex can be either a source or destination node
            mynodes.append(fields[0])
            mynodes.append(fields[2])
            # drop any duplicate vertices
            mynodes = np.ndarray.tolist(np.unique(mynodes))

            # now let's build three lists: sources, destinations and edge_types
            source_nodes.append(fields[0])
            dest_nodes.append(fields[2])
            edge_types.append(fields[1])

    mynode_count=len(mynodes)

    g.add_vertices(mynode_count)

    # populate the vertex properties
    for i in range(0,mynode_count):
        g.vs[i]["id"] = i

```

```

    g.vs["name"] = mynodes[i]
    g.vs[i]["label"] = mynodes[i]

# now let's create the edges
for i in range(0,record_count):
    source=mynodes.index(source_nodes[i])
    dest= mynodes.index(dest_nodes[i])
    type= edge_types[i]
    g.add_edges([(source,dest)])
    g.es[i]["label"] = type
    g.es[i]["name"] = type
    g.es[i]["weight"] = 1

visual_style = {}
# Set bbox and margin
visual_style["bbox"] = (800,800)
visual_style["margin"] = 100

# Set vertex colors and sizes
visual_style["vertex_color"] = 'white'
visual_style["vertex_size"] = 125

# Set vertex label size
visual_style["vertex_label_size"] = 10

# Don't curve the edges
visual_style["edge_curved"] = False

# Set the layout
# my_layout = g.layout_kamada_kawai()
# my_layout = g.layout_circle()
# my_layout = g.layout_drl()
# my_layout = g.layout_fruchterman_reingold()
# lgl = "large graph layout"
# my_layout = g.layout_lgl()
# my_layout = g.layout_random()
# my_layout = g.layout_reingold_tilford()
my_layout = g.layout_reingold_tilford_circular()

visual_style["layout"] = my_layout

# Plot the graph
igraph.plot(g, myfilespec10, **visual_style)

$ cat scrapePage.py
import asyncio
import datetime
import errno
import os
from pathlib import Path
import socket

# https://pypi.org/project/pyppeteer2/
# https://miyakogi.github.io/pyppeteer/reference.html
# One time, download Chromium: $ pyppeteer-install
from pyppeteer import *

import tkinter as tk
from tkinter.messagebox import *
```

```

def display_error():
    tkinter.messagebox.showerror(message=\
        "Couldn't get FQDN.\nTypo in the FQDN entered?",
        title="Message Box", icon="error")

async def scrapeAFQDN(fqdn, my_log_box, url_or_file):
    # the snapshot goes to this filespec
    # if on something non-Un*x-ish, remember os.path.join(dir, f)
    my_log_box.insert(tk.END, "Making sure archive directory exists...\n")
    my_log_box.update()
    home = str(Path.home())
    subdir = "snapshots"
    myfilename = fqdn
    utcdatetime = datetime.datetime.utcnow().isoformat() + 'Z'
    myextension = "jpeg"
    mydir = home + "/" + subdir
    fullfilenamepart = fqdn + "_" + utcdatetime + "." + myextension
    latestversion = fqdn + "_" + "latest" + "." + myextension

    # ensure the directory exists
    try:
        os.makedirs(mydir)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise

    # ensure the timestamped file exists
    if (url_or_file == "url"):
        myfilespec = mydir + "/" + fullfilenamepart
        # ensure the directory exists
        try:
            os.makedirs(mydir)
        except OSError as e:
            if e.errno != errno.EEXIST:
                raise
    elif (url_or_file == "file"):
        myfilespec = fullfilenamepart
    try:
        open(myfilespec, 'a').close()
    except:
        raise

    my_log_box.insert(tk.END, "Confirming page is reachable on 443 or 80...\n")
    my_log_box.update()

    domain_resolve_ok = False
    if (url_or_file == "url"):
        # verify page exists, and figure out if https is supported
        try:
            # note: you do NOT need to explicitly permit inbound traffic
            # on macOS (even if it asks you to allow it!)
            myaddrinfo = socket.getaddrinfo(fqdn, 443)
            url = "https://" + fqdn
            domain_resolves_ok = True
        except socket.gaierror:
            # couldn't connect on default SSL/TLS port
            # fall back to regular http
            try:

```



```

        myaddrinfo = socket.getaddrinfo(fqdn, 80)
        url = "http://" + fqdn
        domain_resolves_ok = True
    except:
        # couldn't even connect on standard port
        display_error()
        return(1)
elif (url_or_file == "file"):
    domain_resolves_ok = True
    url = "file://" + fqdn
    myfilespec = fqdn + ".jpeg"

if domain_resolves_ok == True:
    browser = await launch({'headless': True, 'ignoreHTTPSErrors': True, \
        'defaultViewport': None, 'viewport_width': 800})
    context = await browser.createIncognitoBrowserContext()
    page = await browser.newPage()
    await page._client.send('Animation.disable')
    await page.goto(url)
    await page.screenshot({'path': myfilespec, 'type': 'jpeg', \
        'quality': 80, 'fullPage': True})
    await browser.close()

# ensure the "latest" version is updated for this URL
if (url_or_file == "url"):
    mylatest = mydir + "/" + latestversion
    # print("mylatest =" + mylatest)
elif (url_or_file == "file"):
    mylatest = myfilespec + "_" + "latest" + "." + myextension
try:
    os.unlink(mylatest)
except:
    pass
os.symlink(myfilespec, mylatest)

return(myfilespec)

```

APPENDIX III. STANDARD LIBRARIES AND THIRD-PARTY PACKAGES

- `asyncio`²⁸
- `contextlib`²⁹
- `datetime`³⁰
- `errno`³¹
- `io`³²
- `json`³³
- `os`³⁴
- `pathlib`³⁵
- `re`³⁶
- `socket`³⁷
- `sys`³⁸
- `time`³⁹

- **dnspython:**
 - <https://www.dnspython.org/>
 - <https://dnspython.readthedocs.io/en/latest/index.html>

- **ipwhois** was used for IP Whois queries:
 - <https://pypi.org/project/ipwhois/>
 - <https://ipwhois.readthedocs.io/en/latest/>

- **IP2Location** is our solution for geolocating IP addresses:
 - <https://www.ip2location.com/development-libraries/ip2location/python>
 - <https://www.ip2location.com/free/applications>

- XML processing is with **lxml**:
 - <https://pypi.org/project/lxml/>
 - <https://lxml.de/>

- We also used **numpy** to uniquify some data:
 - <https://pypi.org/project/numpy/>

²⁸ <https://docs.python.org/3/library/asyncio.html>

²⁹ <https://docs.python.org/3/library/contextlib.html>

³⁰ <https://docs.python.org/3/library/datetime.html>

³¹ <https://docs.python.org/3/library/errno.html>

³² <https://docs.python.org/3/library/io.html>

³³ <https://docs.python.org/3/library/json.html>

³⁴ <https://docs.python.org/3/library/os.html>

³⁵ <https://docs.python.org/3/library/pathlib.html>

³⁶ <https://docs.python.org/3/library/re.html>

³⁷ <https://docs.python.org/3/library/socket.html>

³⁸ <https://docs.python.org/3/library/sys.html>

³⁹ <https://docs.python.org/3/library/time.html>

- <https://numpy.org/>
- We'll use **Pillow** (derived from the Python Image Library, or "PIL") to convert and resize images:
 - <https://pypi.org/project/Pillow/>
 - <https://pillow.readthedocs.io/en/stable/>
- **plotly.express** (with **pandas** and **kaleido**) will be used for making maps:
 - <https://pypi.org/project/plotly-express/>
 - <https://plotly.com/python/plotly-express/>
 - <https://pypi.org/project/pandas/>
 - <https://pandas.pydata.org/pandas-docs/stable/>
 - <https://pypi.org/project/kaleido/>
- ASN Prefixes will be obtained via **pyasn**:
 - <https://pypi.org/project/pyasn/>
 - <https://github.com/hadiasghari/pyasn>
- We'll retrieve web pages using **pycurl** and **Requests**:
 - <https://pypi.org/project/pycurl/>
 - <http://pycurl.io/>
 - <https://pypi.org/project/requests/>
 - <https://requests.readthedocs.io/en/master/>
- **PyMuPDF** ("fitz") is the package we'll use for displaying PDFs:
 - <https://pypi.org/project/PyMuPDF/>
 - <https://readthedocs.org/projects/pymupdf/>
- We'll save graphical copies of web sites using **Pyppeteer** (the Python version of Puppeteer):
 - Get the library: <https://pypi.org/project/pyppeteer2/>
 - <https://miyakogi.github.io/pyppeteer/reference.html>
- The network graph gets built with **python-igraph**
 - <https://igraph.org/python/>
- We'll use **Tkinter** (tk) for our GUI environment
 - Tkinter is bundled with Python installed from <https://www.python.org/downloads/>
 - Documentation: <https://docs.python.org/3/library/tk.html>
 - <https://docs.python.org/3/library/tkinter.ttk.html>
- Danny Cork's **whois** is our pick for Domain Whois queries:
 - <https://pypi.org/project/whois/>
 - <https://github.com/DannyCork/python-whois>

APPENDIX IV. LICENSES AND ACKNOWLEDGEMENTS

- **Python:** PSF License
<https://docs.python.org/3/license.html>
- **dnspython:** ISC License
<https://github.com/rthalley/dnspython/blob/master/LICENSE>
- **igraph:** GNU GPL 2 or later License
<https://igraph.org/python/#contribute>
- **ipwhois:** BSD License
<https://pypi.org/project/ipwhois/>
- **IP2Location:** <https://lite.ip2location.com/terms-of-use>
"This site or product includes IP2Location LITE data available from https://lite.ip2location.com."
- **kaleido:** MIT License
<https://pypi.org/project/kaleido/>
- **lxml:** BSD License
<https://pypi.org/project/lxml/>
- **numpy:** BSD License
<https://pypi.org/project/numpy/>
- **pandas:** 3-clause ("Simplified" or "New") BSD license
- **Pillow:** "OSI Approved :: Historical Permission Notice and Disclaimer (HPND)"
<https://pypi.org/project/Pillow/>
- **plotly.express:** MIT License
https://github.com/plotly/plotly_express/blob/master/LICENSE.txt
- **pyasn:** "OSI Approved :: MIT License"
<https://pypi.org/project/pyasn/>
- **pycurl:** "LGPL and an MIT/X derivative license based on the cURL license."
<http://pycurl.io/>
- **PyMuPDF:** "GNU Affero General Public License v3 or later (AGPLv3+), GNU General Public License v3 or later (GPLv3+)"
<https://pypi.org/project/PyMuPDF/>
- **Pyppeteer:** MIT License
- **requests:** Apache2 License
<https://2.python-requests.org/en/v2.7.0/user/intro/>
- **Tkinter:** Python License
<https://en.wikipedia.org/wiki/Tkinter>
- **whois:** MIT License
<https://pypi.org/project/whois/>

APPENDIX V. SCHOOL WEB PAGES REVIEWED FOR HEIGHT MEASUREMENT PURPOSES

School	URL	Image Size	Comments
1. Arizona State University	www.asu.edu	800 x 2070	Captures as a blank page
2. Bowdoin	www.bowdoin.edu	800 x 5696	
3. Brown	www.brown.edu	800 x 8110	
4. BYU	www.byu.edu	800 x 2851	
5. Cal Tech	www.caltech.edu	816 x 4072	
6. Carleton	www.carleton.edu	800 x 3278	Didn't capture cleanly
7. Carnegie Mellon	www.cmu.edu	800 x 5368	
8. Citadel	www.citadel.edu	800 x 4194	
9. Clemson	www.clemson.edu	800 x 8357	Didn't capture cleanly
10. Colby	www.colby.edu	800 x 1711	Didn't capture cleanly
11. Columbia	www.columbia.edu	800 x 7202	
12. Cornell	www.cornell.edu	800 x 3665	
13. Dartmouth	www.dartmouth.edu	800 x 3718	
14. Duke	www.duke.edu	800 x 4962	
15. Eastern Oregon University	www.eou.edu	800 x 4360	
16. Emory	www.emory.edu	800 x 11159	
17. Florida International University	www.fiu.edu	800 x 5827	
18. Georgetown	www.georgetown.edu	800 x 9246	
19. George Washington University	www.gwu.edu	816 x 7093	
20. Georgia Tech	www.gatech.edu	800 x 4529	
21. Harvard	www.harvard.edu	800 x 3467	
22. Harvey Mudd College	www.hmc.edu	800 x 3315	
23. Indiana University Bloomington	www.indiana.edu	800 x 4712	
24. Johns Hopkins	www.jhu.edu	800 x 6804	Didn't capture cleanly
25. Lewis and Clark	www.lclark.edu	1050 x 2261	
26. LSU	www.lsu.edu	850 x 4313	
27. MIT	www.mit.edu	800 x 2263	
28. Montana State University	www.montana.edu	800 x 3379	
29. New York University	www.nyu.edu	800 x 1594	
30. Northwestern	www.northwestern.edu	800 x 7290	
31. Norwich	www.norwich.edu	800 x 5905	
32. Notre Dame	www.nd.edu	800 x 6182	
33. Ohio State University	www.osu.edu	n/a	Failure: Redirection loop
34. Oregon Health and Science University	www.ohsu.edu	800 x 3863	
35. Oregon Institute of Technology	www.oit.edu	800 x 5333	
36. Oregon State University	www.oregonstate.edu	800 x 6903	
37. Penn State	www.psu.edu	800 x 7810	Didn't capture cleanly
38. Pomona	www.pomona.edu	800 x 2023	
39. Portland State	www.pdx.edu	800 x 6471	
40. Princeton	www.princeton.edu	800 x 6312	
41. Reed	www.reed.edu	800 x 5277	
42. Rice	www.rice.edu	800 x 7688	
43. RPI	www.rpi.edu	800 x 5536	
44. Sewanee	www.sewanee.edu	800 x 600	Didn't capture cleanly
45. Southern Oregon University	www.sou.edu	800 x 7412	Didn't capture cleanly
46. St. Olaf	www.stolaf.edu	1600 x 1200	Didn't capture cleanly
47. Stanford	www.stanford.edu	800 x 10769	Didn't capture cleanly

48.	Texas A&M University	www.tamu.edu	800 x 3485	
49.	Tufts	www.tufts.edu	816 x 4813	
50.	West Point	www.westpoint.edu	800 x 7451	
51.	Western Oregon University	www.wou.edu	800 x 6972	
52.	University of Alabama	www.ua.edu	800 x 11071	Didn't capture cleanly
53.	University of Alaska at Fairbanks	www.uaf.edu	800 x 5033	
54.	University of Arizona	www.arizona.edu	800 x 5576	
55.	University of Arkansas	www.uark.edu	800 x 3766	
56.	University of California Berkeley	www.berkeley.edu	800 x 2647	
57.	University of California Los Angeles	www.ucla.edu	800 x 2559	
58.	University of California San Diego	www.ucsd.edu	800 x 8393	
59.	University of Chicago	www.uchicago.edu	800 x 3391	
60.	University of Colorado at Boulder	www.colorado.edu	800 x 6836	
61.	University of Florida	www.ufl.edu	800 x 3472	
62.	University of Georgia	www.uga.edu	800 x 4581	
63.	University of Hawaii-Manoa	www.hawaii.edu	800 x 5013	
64.	University of Idaho	www.uidaho.edu	800 x 3084	
65.	University of Illinois-Urbana Champaign	www.uiuc.edu	800 x 3507	
66.	University of Iowa	www.uiowa.edu	800 x 6772	
67.	University of Kansas	www.ku.edu	800 x 5662	
68.	University of Maryland	www.umd.edu	800 x 3081	Didn't capture cleanly
69.	University of Michigan	www.umich.edu	800 x 5975	
70.	University of Minnesota at Minneapolis	www.umn.edu	800 x 6450	
71.	University of Mississippi	olemiss.edu	800 x 2036	Captures as a blank page
72.	University of Nebraska at Lincoln	www.unl.edu	800 x 5438	
73.	University of North Carolina at Chapel Hill	www.unc.edu	800 x 5598	
74.	University of North Dakota	www.und.edu	800 x 6641	
75.	University of North Georgia	www.ung.edu	800 x 4477	
76.	University of Oklahoma	www.ou.edu	980 x 2017	
77.	University of Oregon	www.uoregon.edu	800 x 11461	
78.	University of Pennsylvania	www.upenn.edu	800 x 3088	
79.	University of Pittsburgh	www.pitt.edu	800 x 4723	
80.	University of Tennessee at Knoxville	www.utk.edu	800 x 3442	
81.	University of Texas at Austin	www.utexas.edu	800 x 11110	Didn't capture cleanly
82.	University of Utah	www.utah.edu	800 x 5617	
83.	University of Virginia	www.virginia.edu	800 x 5033	
84.	University of Washington	www.washington.edu	800 x 2978	
85.	University of Wisconsin at Madison	www.wisc.edu	800 x 4337	
86.	USC	www.usc.edu	800 x 2788	
87.	Vanderbilt	www.vanderbilt.edu	800 x 4680	
88.	Virginia Tech	www.vt.edu	800 x 5569	Overlay issue
89.	VMI	www.vmi.edu	800 x 3325	
90.	Wake Forest	www.wfu.edu	800 x 4757	Didn't capture cleanly
91.	Washington State University	www.wsu.edu	800 x 5993	
92.	Washington University of St Louis	www.wustl.edu	800 x 6688	
93.	Whitman	www.whitman.edu	800 x 3976	
94.	Willamette	www.willamette.edu	800 x 7269	Didn't capture cleanly
95.	Williams	www.williams.edu	800 x 3364	
96.	Yale	www.yale.edu	800 x 3865	



About Farsight Security, Inc.

Farsight Security®, Inc. is the world's largest provider of historical and real-time DNS intelligence solutions. We enable security teams to qualify, enrich and correlate all sources of threat data and ultimately save time when it is most critical - during an attack or investigation. Our solutions provide enterprise, government and security industry personnel and platforms with unmatched global visibility, context and response. Farsight Security is headquartered in San Mateo, California, USA.

Learn more about how we can empower your threat platform and security team with Farsight Security passive DNS solutions at <https://www.farsightsecurity.com> or follow us on Twitter: **@FarsightSecInc.**

To schedule a demo and obtain a free trial, contact: sales@farsightsecurity.com

+1-650-489-7919

Farsight Security®, Inc. 177 Bovet Rd Ste 180 San Mateo, CA 94402 USA
info@farsightsecurity.com www.farsightsecurity.com