Module: Network Communication Over HTTP(S) and Sockets

(b)(3)-P.L. 86-36

0

Updated over 2 years ago by [redacted] in COMP 3321

3 336 71

fcs6 python

(U) Module: Network Communication Over HTTP(S) and Sockets

Recommendations

(U) HTTP with **requests**

(U) There are complicated ways of interacting with the network using built-in libraries, such as **urllib**, **urllib2**, and **httplib**. We'll forgo those in favor of the **requests** library. This is included with Anaconda, but generally not with with other python interpreters. So for this notebook, you'll want to execute it on an Anaconda jupyter-notebook, not in labbench. In general, you can pip install it on other python implementations.

```
$ pip install requests
```

P.L. 86-36

```
import requests
```

```
# One of the few things not yet requiring a certificate for Secure The Net.
resp = requests.get([redacted])
```

```
print(resp.status_code)
print(len(resp.content))
print(len(resp.text))
```

```
# bytes vs. unicode
resp.content == resp.text
```

```
resp.content
```

```
resp.url
```

```
resp.ok
```

```
resp.headers
```

(U) Other HTTP methods, including `put`, `delete`, and `head` are also supported by **requests**

(U) Setting up PKI

(U) Convert P12 certificate to PEM

(U//~~FOUO~~) The `requests` module needs your digital signature certificate to be in PEM format. This section assumes you're starting with a P12 formatted certificate, which is what you commonly start with. If you can't find your P12 cert, you may be able to export it from your browser. If you use CSPid, other instructions may apply. We'll also do this in python below, so you don't have to do this now.

1. (U) Windows Start > type 'cygwin' > run Cygwin Terminal
2. (U//~~FOUO~~) Run `cd /cygdrive/u/private/Certificates` (or whatever directory holds your .p12)
3. (U) Run `openssl pkcs12 -clcerts -in <your DS cert>.p12 -out <your DS cert>.pem`

- (U) Enter your existing certificate password
- (U) Enter a new pass phrase. It's generally a good idea to re-use the .p12 password.
- (U) Confirm the new pass phrase.

(U//~~FOUO~~) Get the CA trust chain

(U//~~FOUO~~) To interact with sites over HTTPS, Python will need to know which certificate authorities to trust. To tell it that, you will need the following file.

1. (U//~~FOUO~~) Visit the PKI certificate authorities page (or "go pki" > click on "CA Chains" under "Server Administrators")
2. (U//~~FOUO~~) Scroll down to "Apache Certification Authority Bundles" at the bottom and click to expand "All Trusted Partners Apache Bundles"
3. (U//~~FOUO~~) Right click on "AllTrustedPartners.crt" and save it into the directory holding your .p12 certificate

(U) HTTPS and PKI with **requests**

(U) To use PKI, you need the proper Certificate Authority_ and PEM-encoded PKI keys. We'll use a `requests.Session` object so that we only have to load these once..

Challenge: find a better algorithym than DES that `dump_privatekey` accepts

```
from OpenSSL import crypto
p12 = crypto.load_pkcs12(open("sid_DS.p12", "rb").read(), b"Your PKI password")
certfile = open("sid_DS.pem","wb")
certfile.write(crypto.dump_privatekey(crypto.FILETYPE_PEM, p12.get_privatekey(), 'DES', b"mypkipassword"))
certfile.write(crypto.dump_certificate(crypto.FILETYPE_PEM, p12.get_certificate()))
certfile.close()
```

```
import requests
```

```
ses = requests.Session()
```

```
ses.verify = 'Apache_Bundle_AllTrustedPartners.crt'
```

```
# Will take the certificate, or a tuple of the certificate and password or at least it used to
# but the current version seems to not want to take a password string
# this avoids us getting prompted for the password
ses.cert = 'sid_DS.pem' #, b"mypkipassword",
```

```
resp = ses.get('https://home.web.nsa.ic.gov/')
```

At this point you need to click over to the terminal running your notebook and respond to the

```
Enter PEM pass phrase:
```

prompt. You should only get one prompt per Session().

```
resp.headers
```

```
resp = ses.get('https://nbgallery.nsa.ic.gov/')
resp.headers
```

P.L. 86-36

(U) It's also easy to POST data to a web service with **requests**:

```
resp = ses.get([redacted])
index1 = resp.text.find('method="post"')
index2 = resp.text.find('</form>',index1)
print (resp.text[index1-64:index2+7])
```

```
payload = {"Name":"[redacted]",
           "sid":"[redacted]",
           "Organization":"[redacted]",
           "others":"[redacted]",
           "message":"Just playing with Python(Jeannie said it was OK)",
           "sendto":"[redacted]",
           "subject":"(U//~~FOUO~~) Testing", "redirect":"",
           "classification":"UNCLASSIFIED~~//FOR OFFICIAL USE ONLY~~"}
```

P.L. 86-36

```
resp = ses.post("https://siteworks.web.nsa.ic.gov/main/emailForm/", data=payload)
```

```
print(resp.text[resp.text.find("Your form"):])
```

(U) In this exmaple, `ses.cert` could also be a list or tuple containing `(certfile, keyfile)`, and `keyfile` can be a password-less PEM file or a PEM file and password string tuple, so you aren't prompted for your password every time.

(U) Low-level socket connections with `socket`

(U) Communication over a socket requires a **server** (which *listens*) and a **client** (which *connects*) to the server, so we'll need to open up two interactive interpreters. Both the server and the client can send and receive data. The **server** must

1. *Bind* to an IP address and port,
2. Announce that it is accepting connections,
3. Listen for connections.
4. Accept a connection.
5. Communicate on the established connection.

We'll run the server (immediately below) in the notebook and the client (below) in a separate python window on the system where we're running our jupyter-notebook.

```
#THIS IS THE SERVER
```

```
import socket
```

```
sock_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # IPv4, TCP
```

```
HOST = '127.0.0.1'
```

```
PORT = 50505 # USE YOUR OWN!
```

```
sock_server.bind((HOST, PORT))
```

```
sock_server.listen(1)
```

```
sock_conn, meta = sock_server.accept()
```

```
sock_conn.send(b"Hello, welcome to the server")
```

```
sock_conn.recv(4096)
```

(U) The **client** must

1. *Connect* to an existing (IP address, port) tuple where a **server** is listening.
2. Communicate on the established connection.

So for our purposes, we'll run the following in a separate python window

```
#THIS IS THE CLIENT
import socket
sock_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST, PORT = '127.0.0.1', 50505 # must match a known server
sock_client.connect((HOST, PORT))
sock_client.recv(512)
sock_client.send(b"Thank you.  I am the client")
```

(U) Buffering, etc. are taken care of for you, mostly.

(U) Topics for Future Consideration:

- SOAP with SOAPpy and/or SUDS
- Using modules from the Standard Library
- XML-RPC
- Parsing HTML with BeautifulSoup

(U)

HTTPS and PKI Concepts

0

Updated over 3 years ago by [redacted]

3 2 575 66

pem pkcs12 p12 pki https pypki2 ssl requests openssl fcs6 ppk

(U) Overview of HTTPS and PKI concepts.

Recommendations

HTTPS and PKI Concepts

PKI is confusing, especially given the mix of internal and external uses, but there are some core concepts.

Public Key Infrastructure (PKI)

Each PKI certificate has two parts, the private key and the public key. The public key is simply an encrypted form of the private key. It is important to keep the private key secret at all costs. A compromized private key would allow someone else to pretend to be the original owner.

Establishing Trust

When you go to amazon.com, your browser receives their server certificate. But how do you know you can trust it?

Certificate Authorities (CA's)

Buried in your browser is a long list of known certificate authorities, such as Verisign. The amazon.com server certificate has been digitally signed by one of these CA's. We know it's coming from amazon.com because only amazon can generate the corresponding public key, and **only the corresponding private key can decrypt traffic sent to the public key**. In other words, because your computer knows the public key is signed by a known CA, and your computer is sending data to that public key, only amazon can decrypt it because they have the corresponding private key.

PKI in the IC

The IC, including NSA, has its own certificate authorities (CA's). Furthermore, both the users and the servers have certificates (generally only servers have certificates on the outside). These certificates are signed by the IC CA's, which are visible at https://pki.web.nsa.ic.gov/pages/certificateAuthorities.shtml

Digital Signature (DS) Certificate

90% of the time, you're using your digital signature certificate. This certificate verifies that you are you to the various services you access on NSAnet. You also use your DS certificate for Secure Shell (SSH) to access systems like MACHINESHOP, LABBENCH, and OpenShift.

Key Encryption (KE) Certificate

On the rare occasion that you encrypt an e-mail, you use your KE certificate. Your browser doesn't actually need this certificate.

Key Formats

PKCS12

NSA keys come in PKCS12 (.p12) format. It contains both the public and private key. With Python, you need the OpenSSL package to use PKCS12 certificates.

PEM

PEM format is by far the most widely supported format on the outside. Many languages and framworks only support PEM, not PKCS12. However, you can convert your key from PKCS12 to PEM format using the openssl command.

To further complicate matters, many languages and frameworks only support **unencrypted** PEM certificates. You can unencrypt your PEM or PKCS12 certificate with the openssl command, but this is generally a no-no since it would allow anyone to masquerade as you.

PPK

PPK format is only used by PuTTY, the SSH tool for Windows. You can convert your key from PKCS12 to PPK format with the P12_to_PPK Converter tool.

PKI with Python

pypki2

Examples at https://gitlab.coi.nsa.ic.gov/python/pypki2/blob/master/README.md

By Hand with ssl Package

SSL is the Secure Sockets Layer, which implements HTTPS (Hyper Text Transfer Protocol Secure)

Python 2.7.9+

```
from getpass import getpass
from urllib2 import build_opener, HTTPCookieProcessor, HTTPError, HTTPSHandler, Request
import ssl

pemPasswd = getpasswd('Enter your PKI password: ')
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.load_cert_chain(pemCertFile, keyfile=pemKeyFile, password=pemPasswd)
context.load_verify_locations(cafile=pemCAFile)
opener = build_opener(HTTPCookieProcessor(), HTTPSHandler(context=context))
req = Request('https://wikipedia.nsa.ic.gov/en/Colossally_abundant_number')
resp = opener.open(req)
print(resp.read())
```

Python 3.4+

```
from getpass import getpass
from urllib.request import build_opener, HTTPCookieProcessor, HTTPSHandler, Request
import ssl

pemPasswd = getpasswd('Enter your PKI password: ')
context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
context.load_cert_chain(pemCertFile, keyfile=pemKeyFile, password=pemPasswd)
context.load_verify_locations(cafile=pemCAFile)
opener = build_opener(HTTPCookieProcessor(), HTTPSHandler(context=context))
req = Request('https://wikipedia.nsa.ic.gov/en/Colossally_abundant_number')
resp = opener.open(req)
print(str(resp.read(), encoding='utf-8'))  # read() returns bytes type, which has to be converted to str type
```

External Packages

OpenSSL

Handles PKCS12 and many other key formats, but not part of the standard library. It is included with Anaconda/Jupyter.

Requests

Supports only unencrypted PEM format. Takes care of a lot of little things for you like HTTP redirects. More on HTTP Status Codes at https://wikipedia.nsa.ic.gov/en/List_of_HTTP_status_codes

Python, HTTPS, and LABBENCH

(b) (3)-P.L. 86-36

0

Updated 3 months ago by [redacted] in COMP 3321

3 10 478 163

fcx91 requestspki labbench

(U//~~FOUO~~) This notebook demonstrates how to interact with web resources over HTTPS when using LABBENCH. It primarily uses the requests_pki module.

Recommendations

UNCLASSIFIED//~~FOR OFFICIAL USE ONLY~~

(U//~~FOUO~~) Python, HTTPS, and LABBENCH

(U//~~FOUO~~) This notebook demonstrates how to interact with web resources over HTTPS when using LABBENCH. It uses the **requests_pki** and **rest_api** modules.

(U) HTTP with **requests_pki**

(U) There are complicated ways of interacting with the network using built-in libraries, such as **urllib**, **urllib2**, and **httplib**. For basic (unsecured) interaction, we can use **requests**. However, with *Secure The Net*, almost everything is now PKI-enabled.

(U) Luckily, there is a module for that! LABBENCH has native support for **requests_pki**, which makes it an ideal library for us.

```
!pip3 install requests
```

```
import ipydeps

modules = ['requests-pki','pypac']

ipydeps.pip(modules)
```

(U) Example 1: Obligatory example of **requests**

```
import requests
# One of the few things not yet requiring a certificate for Secure The Net.
resp = requests.get('http://airs.s2.org.nsa.ic.gov/')
```

```
print("Response code:     {}".format(resp.status_code))
print("Length of content: {}".format(len(resp.content)))
print("Length of text:    {}".format(len(resp.text)))
```

(U) That's interesting. Are **content** and **text** the same?

```
resp.content == resp.text
```

(U) It turns out that **content** stores the *bytes* of the response and **text** stores the *unicode* of the response. Let's look at the text:

```
print(resp.text)
```

(U) That's great if we want the raw HTML...which in many cases we may. However, we can render the HTML response natively within Jupyter!

```
from IPython.display import display, HTML
display(HTML(resp.text))
```

(U) Notice that we didn't get any of the images that go along with this webpage, but for our purposes now this is sufficient.

(U) **requests_pki**

(U//~~FOUO~~) LABBENCH has made interacting with secure webpages trivial! That's because the **requests_pki** module works seamlessly with LABBENCH to pass your PKI with your request. Let's see how easy it is!

(U) Example 2: nbGallery

```
import requests_pki
sess = requests_pki.Session()
```

```
resp = sess.get('https://nbgallery.nsa.ic.gov/')
```

```
resp.headers
```

```
display(HTML(resp.text))
```

(U) So maybe Jupyter isn't meant to be a full-fledged web-browser after all...

(U) Example 3: Notebook Gallery search

(U) Search the Notebook Gallery for a term, get the results back as JSON, and parse the JSON. This adds a new `headers` argument to the GET request.

(U//~~FOUO~~) Normally a web server will respond with some default type of output. That may be `application/html`, `application/xml`, or something else. If you don't like that, you can try persuading the server to give you something else using an `Accept` header. That will tell the server your preferred response format (i.e. the format you prefer to accept). Servers often support multiple formats, but not all of them.

```
import json
```

```
search_term = 'beautifulsoup'
url = "https://nbgallery.nsa.ic.gov/notebooks"
params = { 'q' : search_term, 'sort' : 'score' }
headers = { 'Accept' : 'application/json' }
resp = sess.get(url, params=params, headers=headers)
```

```
resp.url
```

```
print(resp.text)
```

```
# json.loads() will parse a JSON string into lists and hashes
resp_parsed = json.loads(resp.text)
```

```
type(resp_parsed)
```

```
# take a look at it and find what you want
resp_parsed
```

```
# print the titles of all notebooks that matched your search term
[ record['title'] for record in resp_parsed ]
```


(U) Example 4: Using a proxy

(U) Sometimes you need a proxy set up, particularly when working with second party sites. **requests_pki** and **pypac** make this setup quite easy!

```
import pypac

proxy = 'http://www.web.nsa.ic.gov/proxy/ipsec.pac'
url = '[redacted]
sess = requests_pki.Session(pac=proxy)
params = { 'type' : 'Community' , 'activity_area' : 'All' , 'project' : 'All' , 'service' : 'All' }
respFromCSE = sess.get(url, params=params)
```

```
display(HTML(respFromCSE.text))
```

(U) Example 5: Post with JSON

Sometimes you'll need to 'post' data rather than do a 'get' request. The 'post' works similar to the 'get', but you'll need to specify parameters for the post and usually need to set the headers as well. This one posts the parameters as a JSON object; another common content type is `application/x-www-form-url-encoded` , in which you'll need to use the `urllib` library to URL encode your parameters prior to posting them.

```
base_url = 'https://namingstuff-mestern.apps.oso4.platform.cloud.nsa.ic.gov/'

# with this post, we're telling the host that we are sending json, and want to receive json
# the post parameters are sent in the 'data' key, and must be json in this case
status_code = 0
tries = 0
while not status_code == 200:
    resp = sess.post(
      base_url + 'GetRecord/languages/languages',
      headers={'Accept': 'application/json', 'Content-Type': 'application/json'},
      data=json.dumps({'language': {'$ne': 'English'}})
    )
    status_code = resp.status_code
    tries += 1
    if tries > 3:
        break
```

```
print(resp.status_code)
languages = json.loads(resp.text)
print(len(languages))
print(languages[0])
```

(U) rest_api

The `rest_api` library is another resource for accessing HTTPS pages on NSANet. Like `requests_pki`, `rest_api` takes care of all the PKI authentication for you, but this library is built to enable you to create what's called an 'API wrapper', which means that we're wrapping our own class around the API, which is designed to just make it easier to query the API and interpret the results. API, by the way, stands for Application Programming Interface, and is basically a clearly defined set of methods for communication with a given service, or rules for interacting with data housed in a web service.

In general if you want to hit a single web page, `requests_pki` is generally preferred because there's less overhead (you don't have to create a whole class to do it). But if you want to hit multiple pages at a website or API, then `rest_api` is probably the better way to go.

(U) Example 6: rest_api with TESTFLIGHT

This example shows a simple class that inherits from `rest_api.AbstractRestAPI`, and allows us to hit a couple of pages (called 'endpoints') of the TESTFLIGHT API. Notice we set `host` and `headers` as class variables. With these set, we don't have to define them every time we make a query to a TESTFLIGHT page. For each page we just add the actual page or endpoint and the class fills in the rest of the URL.

```
ipydeps.pip('rest-api')
import rest_api
```

```
class Testflight(rest_api.AbstractRestAPI):
    host = 'https://tf-www.testflight.proj.nsa.ic.gov'
    headers = {'Accept': 'application/json'}

    def sources(self):
        "Returns a list of all sources that feed Testflight"
        endpoint = '/SolanoService/rest/report/sources'
        return self._get(endpoint).json()

    def search(self, **kwargs):
        "Returns report summaries that match the given keyword arguments"
        endpoint = '/SolanoService/rest/report/search/'
        return self._post(endpoint, data=kwargs).json()
```

```
from pprint import pprint
tf = Testflight()
pprint(tf.sources()[:3])
```

```
pprint(tf.search(originator='NSA', fields="subject serial nipf", start=0, rows=3, sort='Newest'))
```

(U) Other resources

- (U//~~FOUO~~) Other notebooks on the Notebook Gallery that use `requests` (can you modify example 4 above to find them?)
- (U//~~FOUO~~) pypki2, an open source module for working with your P12 certificate that originated at NSA. It's not part of Anaconda and works best in Jupyter on LABBENCH. It works with `urllib.requests` instead.

(U) One more comment. Be careful when you try to display the HTML from webpages...some webpages may affect things more than you want...

```
resp = sess.get('https://home.web.nsa.ic.gov/')
display(HTML(resp.text))
```

Module HTML Processing With BeautifulSoup

0

(b)(3)-P.L. 86-36

Updated 9 months ago by [redacted]

3 326 78

fcx92 beautifulsoup

(U) BeautifulSoup module for COMP3321.

Recommendations

(U) BeautifulSoup is a Python module designed to help you easily locate and pull information out of an HTML document (or string).

(U) A good deal of the time, maybe even the majority of the time, when you have to get your data from the interwebs you will query a web service that returns complete, well formatted responses (JSON, XML, etc). However, sometimes you just have to deal with the fact that the data you want can only be obtained by parsing a messy, probably automatically generated, web page.

(U) There are several approaches to dealing with web page parsing, and several Python packages that can help you. In this lesson we cover one of the most common, BeautifulSoup.

(U//~~FOUO~~) If you are running this via Jupyter on Anaconda, you can import BeautifulSoup and use the requests module to do the [redacted] example. If you want to perform the [redacted] homepage example on Anaconda, you will need to export your signature.PKI to PEM format (instructions here) and use a module that supports HTTPS such as urllib.request.

P.L. 86-36

(U//~~FOUO~~) If you are running Jupyter on LABBENCH, execute the below cell to install bs4 and rest_api

```
import ipydeps

modules = ['bs4','rest_api']

ipydeps.pip(modules)

import rest_api
```

(U) Run this cell regardless of LABBENCH of Anaconda

```
import requests
from bs4 import BeautifulSoup
from IPython.display import HTML, display
```

(~~C~~) Let's grab the [redacted] homepage and save the table on the page as nice, parseable (is that a word?) text. Notice that we can do a simple .get() from the requests module. This is because the [redacted] homepage is one of the very few plain HTTP sites left on the high side.

Doc ID: 6689695

```
# We will try to connect and catch any exceptions in case things go awry
try:
  resp = requests.get('[redacted]')
except:
  print("Well, that didn't work!")

#uncomment these lines if you want to see some of the helpful attributes of the response object

# print(resp.status_code)
# print(len(resp.content))
# print(len(resp.text))

#If we got this far then we have a response (we are going to assume the response isn't
# "Access Denied") we take the response text and create a BeautifulSoup object so we can
# tiptoe through our data
bsObj = BeautifulSoup(resp.text, "html.parser")

#Also could have used bsObj.find_all, this returns a list of all the <table>'s in the HTML
tables = bsObj.findAll("table")

#open our output file for writing
outfile = open([redacted]_table.txt','w')

#loop through our list of tables from the findAll("table") above and go through the table
# one row (<tr>) and cell (<td>) at a time, outputting the information to the screen as csv
# and to the output file in pipe('|') delimited formats.
for table in tables:
  i = 0
  for tr in table.findAll('tr'):
    i += 1
    j = 0
    for td in tr.findAll('td'):
      print('{},'.format(td.text),end='')
      outfile.write("element{}:{}|".format(j,td.text))
      j += 1
    outfile.write('\n')
    print()
outfile.close()
```

P.L. 86-36

(U) Notice how we can display a hyperlink to our output - this might be handy if you don't want to go to Jupyter Home to display the file.

```
display(HTML('<a href="{}" target="_blank">display file</a>'.format("[redacted]table.txt")))
```

(U) Now lets try something a little trickier. Let's pull down the [redacted] homepage and redisplay the "Current Activities" bulleted list inline in our notebook.

P.L. 86-36

```
#We are going to use the rest_api module here.  This is a NSA specific package and has
# HTTPS support baked in. It makes pulling webpages using your PKIs a snap, even though
# the package was really designed to access RESTful webservices and not web pages.
urlString = [redacted]

parameters = ''
headers = { 'text/html',
            'application/xhtml+xml',
            'application/xml;q=0.9',
            '*/*;q=0.8'
          }
queryString = ""
#Create an api object for our host server
api = rest_api.AbstractRestAPI(host=urlString)

#Get the homepage from the server. If you wanted sub-pages off the server you would put
# that path in the queryString as something like "/folder/page.html".
try:
  resp = api._get(queryString)
except:
  print("Well that didn't work!")

#Create our BeautifulSoup object
bsObj = BeautifulSoup(resp.text, "html.parser")
```

```
#Use the .find method to get the <body> of the HTML document. We will drill down to our
# list from there. .find() only returns the first matching HTML tag, which is OK in this
# case because you *should* only have one <body> tag in the document
body = bsObj.find('body')
```

(U//~~FOUO~~) Now for the sticky bit.

From the the Chrome brower Tools-> Developer Tools console (could have done this in Firefox as well from Tools->Web Developer->Toggle Tools) I ascertained that the path through the HTML to the bulleted list I care about is

section2 > div.item-container.item-container2.item-container-rss.item147067 > div.item-content.item-content2 > div > div

more succinctly, as xpath it is

//*[@id="section2"]/div[2]/div[1]/div/div/

but BeautifulSoup does not accept xpath (whomp, whomp). If you like to use xpath the lxml module does a decent job of parsing HTML and does accept xpath syntax.

```
#Now I progress through the body object using the find_next method to get to the bulleted list
activities = body.find_next('div',{'id':'section2'}).find_next('div',{'class':'feedDisplay'})
```

(U) Now we have the right element in the activities object. We can use the **str**() method to get the raw HTML from the object and either print it inline in the notebook or we can just print the text using the .text attribute.

```
# print(activities)
display(HTML(activities.__str__()))
```

An easier way: using 'select'

Doc ID: 6689695

In the 'Inspector' view in your Developer Tools, you can right-click on your desired tag and choose 'Copy Unique Selector' to copy the CSS selector path for your tag. Then you can use `soup.select` or `soup.select_one` to navigate directly to that tag, rather than crawling through the entire hierarchy to get to it. (Note: I ran this in Firefox, not sure what the right-click menu is like in Chrome)

```
selector = ".rssEntries > li:nth-child(1) > div:nth-child(3)"
# at least for our version of bs4, you have to replace
# nth-child with nth-of-type
selector = selector.replace("nth-child", "nth-of-type")
# bsObj.select would find all tags with that path
bsObj.select_one(selector)
```

Module: Operations with Compression and Archives

(b)(3)-P.L. 86-36

0

Updated about 2 years ago by [redacted] in COMP 3321

3 70 8

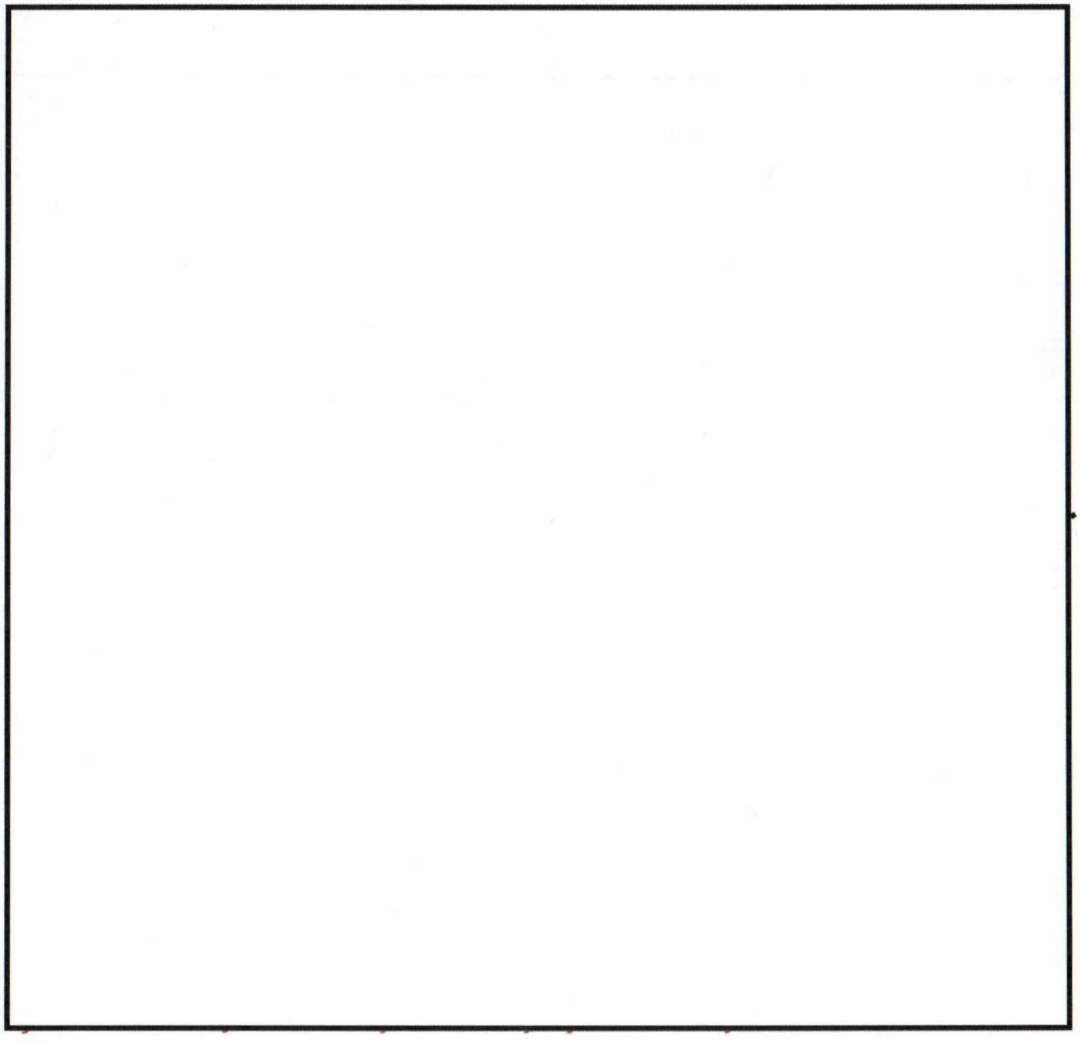
fcs6 python

(U) Module: Operations with Compression and Archives

Recommendations

Doc ID: 6689695

```
user_string = '''
name,username,city,state,zip_code,primary_workstation
```



P.L. 86-36

```
'''

json_string = '''
[{"author": "Jane Austen", "title": "Pride and Prejudice"}, {"author": "Fyodor Dostoevsky", "title": "Crime and Punishment"}, {
'''

with open('user_file.csv','w') as f:
  f.write(user_string)
```

```
with open('user_file.json', 'w') as f:
    f.write(json_string)
```

zipfile

```
import zipfile
```

```
with zipfile.ZipFile('user_file.zip', mode='w') as zf:
   zf.write('user_file.csv')
```

```
zf = zipfile.ZipFile('user_file.zip') # with a filename
```

```
zf2 = zipfile.ZipFile(open('user_file.zip')) # with a file or file-like object
zf2 == zf
```

```
zf.namelist()
```

```
zf2.namelist()
```

```
z = zf.filelist[0]
z
```

```
z.filename, z.file_size
```

```
[(z.filename, z.file_size) for z in zf.filelist]
```

```
zf.getinfo('user_file.csv')
```

```
user_file_csv = zf.open('user_file.csv','r') # returns a file-like object!
```

```
from csv import DictReader
user_data = [_ for _ in DictReader(user_file_csv)]
print(len(user_data))
user_data[0]
```

```
user_file_csv.read()
```

```
user_file_csv.close()
```

```
zf.extract(zf.filelist[0],'zfextract')
```

gzip

```
import gzip
```

```
with gzip.open('user_file.csv.gz', 'wt') as gf:
    gf.write('This string will be stored as text')
```

```
gzip_users = gzip.open('user_file.csv.gz') # takes a file name, returns a file-like object!
x = gzip_users.readlines()
gzip_users.close()
x[:3]
```

```
gzip_users = gzip.open('user_file.csv.gz', 'rt')
g_user_dicts = list(DictReader(gzip_users))
g_user_dicts[:2]
```

```
with open('user_file.csv.gz','rb') as f:
    still_gzipped = f.read()
still_gzipped[:100]
```

```
from io import StringIO
unpacked_users = gzip.GzipFile(fileobj=io.StringIO(still_gzipped)) # what if you have bytes or a file-like obejct to unpack?
unpacked_users.readlines()[:3]
```

tarfile

```
import tarfile
```

```
with tarfile.open('userfile.tar', mode='w') as tf:
    tf.add('user_file.csv')
    tf.add('user_file.json')
```

```
tarfile.is_tarfile('userfile.tar'), tarfile.is_tarfile('user_file.csv')
```

```
tf = tarfile.open('userfile.tar') # don't need to unzip first!
```

```
tf.getmembers()
```

```
tf.getnames()
```

Doc ID: 6689695

```
u = tf.extractfile('user_file.csv')
u2 = tf.extractfile(tf.getmembers()[1])
```

```
u.readline()
```

```
u2.read()[:150]
```

```
tf.extractall('from_tarball')
```

Module: Regular Expressions

(b)(3)-P.L. 86-36

0

Updated 11 months ago by [redacted] in COMP 3321

3 322 154

fcs6 python comp3321

(U) Module: Regular Expressions

Recommendations

(U) Regular Expressions (Regex)

(U) Now You've Got Two Problems...

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
> Jamie Zawinski, 1997

(U) A **regular expression** is a tool for finding and capturing patterns in text strings. It is very powerful and can be very complicated; the *second problem* referred to in the quote is a commentary on how regular expressions are essentially a separate programming language. As a rule of thumb, use the `in` operator or string methods like `find` or `startswith` if they are suitable for the task. When things get more complicated, use regular expressions, but try to use them sparingly, like a seasoning. At times it may be tempting to write one giant, powerful, super regular expression, but that is probably not the best thing to do.

(U) The power of regular expressions is found in the special characters. Some, like `^` and `$`, are roughly equivalent to string methods `startswith` and `endswith`, while others are more flexibile, especially `.` and `*`, which allow flexible matching.

(U) Getting Stuff Done without Regex

```
"mike" in "so many mikes!"
```

```
"mike".startswith("mi")
```

```
"mike".endswith("ke")
```

```
"mike".find("k")
```

```
"mike".isalpha()
```

```
"mike".isdigit()
```

```
"mike".replace("k", "c")
```

(U) Regular expressions in Python

There are only a few common methods for using the **re** module, but they don't always do what you would first expect. Some functionality is exposed through *flags*, which are actually constants (i.e. `int` defined for the **re** module), which means that they can be combined by addition.

```
import re
```

```
re.match("c", "abcdef")
```

```
re.match("a", "abcdef")
```

```
re.search("c", "abcdef")
```

```
re.search("C", "abcdef")
```

```
re.search("C", "abcdef", re.I) # re.IGNORECASE
```

```
re.search("^c", "ab\ncdef")
```

```
re.search("^c", "ab\ncdef",re.M) # re.MULTILINE
```

```
re.search("^C", "ab\ncdef", re.M + re.I)
```

(U) In both `match` and `search`, the *regular expression* precedes the string to search. The difference between the two functions is that `match` works only at the beginning of the string, while `search` examines the whole string.

(U) When repeatedly using the same regular expression, *compiling* it can speed up processing. After a compiled regular expression is created, `find`, `search`, and other methods can be called on it, and given only the search string as a single argument.

```
c_re = re.compile("c")
```

```
c_re.search("abcde")
```

Regex Operators

```
. - matches a single character
^ - matches beginning of a string or newline
$ - matches end of string
* - 0 or more of something
+ - 1 or more of something
? - 0 or 1 of something
*?, +?, ?? - don't be greedy (see example below)
{3} - match 3 of something
{2,4} - match 2 to 4 of something
\ - escape character
[lrnLRN] - match any ONE of the letters l, r, n, L, R, N
[a-m] - match any ONE of letters from a to m
[a|m] - match letter a or m
\w - match a letter
\s - match a space
\d - match a digit
```

```
re.search("\w*s$", "Mike likes cheese\nand Mike likes bees")
```

```
re.findall("\(\d{3}\)\s\d{3}-\d{4}", "Hello, I am a very bad terrorist. If you wanted to know, my phone number is (303) 555-2
```

```
re.findall("mi.*ke", "i am looking for mike and not all this stuff in between mike")
```

```
re.findall("mi.*?ke", "i am looking for mike and not all this stuff in between mike")
```

Capture Groups

Put what you want to pull out of the strings in parentheses ()

```
my_string = "python is the best language for doing 'pro'gramming"
result = re.findall("'(\w+)", my_string)
print(result)
print(result[0])
```

Matches and Groups

(U) The return value from a successful call of `match` or `search` is a *match object*; an unsuccessful call returns `None`. First, this is suitable for use in `if` statements, such as `if c_re.search("abcde"): ...`. For complicated regular expressions, the match object has all the details about the substring that was matched, as well as any captured groups, i.e. regions surrounded by parentheses in the regular expression. These are available via the `group` and `groups` methods. Group 0 is always the whole matching string, after which remaining groups (which can be nested) are ordered according to the opening parenthesis.

```
m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
```

```
m.group()
```

```
m.group(1)
```

```
m.group(2)
```

```
m.groups()
```

Other Methods

(U) Other regular expression methods work through all matches in the string, although what is returned is not always straightforward, especially when captured groups are involved. We demonstrate out some basic uses without captured groups. When doing more complicated things, please remember: be careful, read the documentation, and do experiments to test!

```
re.findall("a.c","abcadcaecafc") # returns list of strings
```

```
re.finditer("a.c","abcadcaecafc") # returns iterator of match objects
```

```
re.split("a.", "abcadcaecafc") # returns list of strings.
```

(U) The `sub` method returns a modified copy of the target string. The first argument is the regular expression to match, the second argument is what to replace it with–which can be another string or a function, and the third argument is the string on which the substitutions are to be carried out. If the `sub` method is passed a function, the function should take a single match object as an argument and return a string. For some cases, if

the substitution needs to reference captured groups from the regular expression, it can do so using the syntax `\g<number>`, which is the same as accessing the `groups` method within a function.

```
re.sub("a.*?c", "a--c", "abracadabra")
```

```
re.sub("a(.*?)c", "a\g<1>\g<1>c", "abracadabra")
```

```
def reverse_first_group(matchobj):
    match = matchobj.group()
    rev_group = matchobj.group(1)[::-1]
    return match[:matchobj.start(1)] + rev_group + match[matchobj.end(1):]
```

```
re.sub("a(.*?)c",reverse_first_group,"abracadabra")
```

(U) In the above, we used `start` and `end`, which are methods on a match object that take a single numeric argument–the group number–and return the starting and ending indices in the string of the captured group.

(U) One final warning: if a group can be captured more than once, for instance when its definition is followed by a `+` or a `*`, then only the last occurrence of the group will be captured and stored.

Hashes

0

(b)(3)-P.L. 86-36

Updated 6 months ago by [redacted] in COMP 3321

3 22 4

comp3321

(U) Computing Hashes in Python

Recommendations

(U) Hashes

(U) Let's start with hashes. Hashes map data of arbitrary size to data of fixed size and have a variety of uses:

- securely storing passwords
- verifying file integrity
- efficiently determining if data is the same

(U) There are many different hashing algorithms. You've probably heard of some of the more common ones, such as MD5, SHA1, and SHA256.

(U) Hashes have some useful features:

- they are one-way, meaning that given a hash, there isn't a function to convert it back to the original data
- they map data to a fixed output, which is useful when comparing large amounts of data (such as files)

(U) So let's generate a hash.

```
from hashlib import sha256
sha256('abc'.encode('ascii')).hexdigest()
```

(U) or

```
sha256(b'abc').hexdigest()
```

(U) We all know storing plaintext passwords is bad. A common technique of avoiding this is to store the hash of the password, then check if the hashes match. So we can create a short function to check if the typed password matches the stored hash:

```
def check_password(clear_password, password_hash):
    return sha256(clear_password).hexdigest() == password_hash
```

(U) Does anyone know why storing the hash of a password is bad?

(U) If the password hash database was ever compromised, it would be vulnerable to a pre-computation attack (rainbow table), where an attacker pre-computes hashes of common passwords. There are tools such as scrypt to help mitigate this vulnerability.

(U) How about a safer use of hashes? Suppose you need to look for duplicate files? Doing a byte-per-byte comparison of every file to every other file would be really expensive. A better approach is to compute the hash of each file, then compare the hashes.

```
import os
from hashlib import md5
def get_file_checksum(filename):
    h = md5()
    chunk_size = 8198
    with open(filename, 'rb') as f:
        while True:
            chunk = f.read(chunk_size)
            if len(chunk) == 0:
                break
            h.update(chunk)
    return h.hexdigest()
```

(U) There is a small danger with this approach: collisions. Since we're mapping a lot of data to a smaller amount of data, there is the possibility that two files will map to the same hash. For SHA256, the chances that two files have the same hash are 1 in 2^256, or about 1 in 1.16e+77. So even with a lot of files, the chance of a collision is small.

(U) Notice that we don't need to read in the entire file at once. One really cool feature of hashes is they can be updated:

```
h = sha256(b'abc')
h.update(b'def')
h.hexdigest()
```

```
sha256(b'abcdef').hexdigest()
```

Module: SQL and Python

(b)(3)-P.L. 86-36

0

Updated almost 2 years ago by [redacted] in COMP 3321

3 1 359 163

fcs6 python

(U) Module: SQL and Python

Recommendations

(U) The Odd Couple: Programming and Databases

(U) It makes a lot of sense to keep your data in a database, and programming logic in a program. Therefore, it's worth overcoming the fundamental impedance mismatch between the two technologies. In the most common use cases, where the program isn't too terribly complicated and the data isn't too crazily interconnected, things usually work just fine.

(U) Python has a recommended Database API, although there are slight variations in the way this API is implemented, which is one reason to use a metalibrary like **SQLAlchemy** (we'll get to this later). The standard library only provides an implementation for **SQLite**, in the `sqlite3` package. Connections to other database types require external packages, such as `MySQLdb` (confusingly, to get this you have to `pip install MySQL-python`).

 bobby drop tables

(U) Basics with **sqlite3**

To interact with a database, a program must

1. Establish a connection
2. Create a cursor
3. Execute commands
    - Read the results
    - Commit the changes

4. Close the cursor and/or connection

(U) Using a basic adapter, commands are executed by passing strings containing SQL commands as arguments.

```
import sqlite3
```

```
conn = sqlite3.connect('test.db') # SQLite specific: creates db if necessary
```

```
cur = conn.cursor()
```

```
cur.execute("""create table fruit (
            id integer primary key,
            name text not null,
            color text default "RED"
            )""")
           )
```

```
cur.execute(''' insert into fruit (name) values ("apple")''') # not there yet
```

```
conn.commit() # to make sure it's written
```

```
cur.execute(""" select * from fruit """) # returns the cursor--no need to capture it.
```

```
cur.fetchone()
```

(U) When making changes to the database, it's best to use *parameter substitution* instead of *string subtitution* to automatically protect against unsanitized input. The **sqlite3** module uses `?` as its substitution placeholder, but this differs between database modules (which is a major headache when writing code that might have to to connect to more than one type of database).

```
fruit_data = [('banana','yellow'),
              ('cranberry','crimson'),
              ('date','brown'),
              ('eggplant','purple'),
              ('fig','orange'),
              ('grape','purple')]
```

```
for f in fruit_data:
    cur.execute("""insert into fruit (name, color) values (?,?)""", f)
```

```
cur.execute("""select * from fruit""") # DANGER! DATA HASN'T BEEN WRITTEN YET!
```

```
cur.fetchone()
```

```
cur.fetchmany(3)
```

```
cur.fetchall()
```

(U) A cursor is iterable:

```
more_fruit = [('honeydew','green'),('ice cream bean','brown'),('jujube','red')]
```

```
cur.executemany(""" insert into fruit (name, color) values (?,?)""",more_fruit)
```

```
cur.execute("""select * from fruit""")
```

```
[item[1] for item in cur] # read the name
```

```
cur.execute('PRAGMA table_info(fruit)')
for line in cur:
  print(line)
```

```
cur.fetchall()
```

```
conn.commit() # always remember to commit!
```

(U) In **sqlite3**, many of the methods associated with a `cursor` have shortcuts at the level of a `connection` –behind the scenes, the module creates a temporary cursor to perform the operations. We will not cover it because it isn't portable.

(U) Other Drivers

(U) The most common databases are MySQL and Postgres. Installing the packages to interact with them is often frustrating, because they have non-Python dependencies. Even worse, the most current version of `mysql-python` in PYPI is broken, so we request a different version:

```
(VENV)[                ~]$ pip install mysql-python==1.2.3
(VENV)[                ~]$ pip install oursql
(VENV)[                ~]$ pip install psycopg2
...
Error: pg_config executable not found.
...
```

P.L. 86-36

(U) With enough exceptions to make life very frustrating, they work like `sqlite3` .

(U) SQLAlchemy

(U) SQLAlchemy is a very powerful, very complicated package that provides abstraction layers over interaction with SQL databases. It includes all kinds of useful features like connection pooling. We'll discuss two basic use cases; in both of which we just want to use it to get data in and out of Python.

(U) Cross-Database SQL

(U) Imagine the following scenario: during development you'd like to use SQLite, even though your production database is MySQL. You don't plan to do anything fancy; you already know the SQL statements you want to execute (although there are a couple of things you always wished `sqlite3` would do for you, like returning a `dict` instead of a `tuple`.

(U) Enter SQLAlchemy. It does require that you have a driver installed, e.g. **MySQLdb**, to actually talk to the database, but it takes care of all the ticky-tack syntax details. By default, it even commits changes automatically!

```
import ipydeps
ipydeps.pip('sqlalchemy')
```

```
import sqlalchemy
```

```
engine = sqlalchemy.create_engine('sqlite:///test.db') # database protocol and URL
```

```
result = engine.execute('select * from fruit')
```

```
ans = result.fetchall()
```

```
first_ans = ans[0]
```

```
type(first_ans)
```

```
first_ans[1]
```

```
first_ans.keys()
```

```
first_ans.values()
```

```
engine.execute('''insert into fruit (name) values (?)''',('kumquat'))
```

```
engine.execute('''insert into fruit (name,color) values (?, ?)''',[('lime','green'),('mango','green')])
```

```
result = engine.execute('select * from fruit')
```

```
result.fetchall()
```

(U) Now, to move to MySQL, all you have to do is use a different URL, which follows the pattern:
`dialect+driver://username:password@host:port/database`
The SQLAlchemy documentation lists all the databases and drivers.

(U) As Object Relational Mapper

(U) The real power in SQLALchemy is in using it to store and retrieve Python objects from a database without ever writing a single line of SQL. It takes a little bit of what looks like voodoo at first. We'll skip most of the details for now, at the risk of this being a complete cargo cult activity. Open up a new file called `sql_fruit.py` and put the following into it:

```
from sqlalchemy import create_engine, Column, Integer, String, Date
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///test.db')
Base = declarative_base()
Session = sessionmaker(bind=engine)
db_session = Session()

class Fruit(Base):
    __tablename__ = 'fruit'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    color=Column(String, default="RED")

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def __repr__(self):
        return "<Fruit {}: {}, {}>".format(self.id, self.name, self.color)
```

(U) Now, in the interactive interpreter:

```
from sql_fruit import *

f_query = db_session.query(Fruit)
```

Doc ID: 6689695

```
f_query.all()
```

```
f_query.first()
```

```
nectarine = Fruit('nectarine','orangered')
```

```
db_session.add(nectarine)
```

```
db_session.commit()
```

Easy Databases with sqlite3

0

Created over 3 years ago by [redacted] in COMP 3321

3 1 199 57

fcs6 comp3321 sqlite3 sql average

(U) Example on using sqlite3 to group and average data instead of using dictionaries.

Recommendations

Easy Databases with sqlite3

The great thing about sqlite3 is that it allows you to create a simple, local database without having to install any servers or other tools. The entire database is contained in a single file.

Here we're going to create a simple database that holds daily stock data. This is related to the Structured Data and Dates Exercise from COMP3321 at https://jupyter-gallery.platform.cloud.nsa.ic.gov/nb/884fbd2f/Structured-Data-and-Dates-Exercise

We'll use the same AAPL stock data from Yahoo Finance available at https://urn.nsa.ic.gov/t/0grli

First we import the packages for reading the CSV, parsing the dates, and working with sqlite3.

```
from csv import DictReader
from datetime import datetime
import sqlite3
```

Create the Table

Here we have a function that creates a stocks table in our database (referenced by db_conn). The columns are:

- symbol: Simply holds the stock ticker symbol so we can store records for more than just AAPL.
- year, month, day: We break the date out into three integer columns because it's easier to do queries against precise dates or groups/ranges of dates. If the date were kept as a string, forming the query string would be much more difficult.