

# Course Objectives

After completing this course, students will be able to:

* Summarize the CTE squad's responsibilities, objectives, and deliverables from each CPT stage
* Analyze threat information
* Develop a Threat Emulation Plan (TEP)
* Generate mitigative and preemptive recommendations for local defenders
* Develop mission reporting
* Conduct participative operations
* Conduct reconnaissance
* Analyze network logs for offensive and defensive measures

Course Objectives (Continued)

Students will also be able to:

* Analyze network traffic and tunneling protocols for offensive and defensive measures
* Plan non-participative operations using commonly used tools, techniques and procedures (TTPs)

# Module 2: Threat Emulation (Objectives)

* Conduct reconnaissance
* Generate mission reports from non-participative operations
* Plan a non-participative operation using social engineering
* Plan a non-participative operation using Metasploit
* Analyze network logs for offensive and defensive measures
* Analyze network traffic and tunneling protocols for offensive and defensive measures
* Plan a non-participative operation using Python
* Develop fuzzing scripts
* Develop buffer overflow exploits 

Module 2 — Lesson 1 1: Python Modules (Objectives)

* Conduct active reconnaissance using Python
* Plan exploitation using Python
* Interpret Python module documentation
* Manipulate web content with Python

# Refresher

* Yesterday, the main takeaways were:
* Dangerous Functions
* File Handling
* Introducing Python Modules & the socket library
* Exercise to fetch banner and determine the software and version number of an accessible FTP server

I

Lesson Overview

In this lesson we will discuss:

* Exception Handling
* Standard Libraries Tour  urllib / requests  re / BeautifulSoup  base64

Tidying previous code...

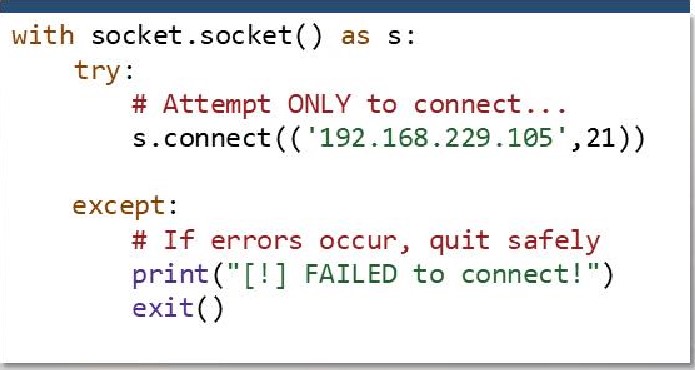
* The last Python script you wrote would connect to a FTP server.
* What if the server is down, or cannot be reached?

|  |
| --- |
| root@kali python3 banner\_grab. py Traceback (most recent call last) :  File "banner\_grab.py", line 6, in <module> '192.168.229.105' , 21))  ConnectionRefusedError: [Errno 111] Connection refused |
|
|
|  |

* Before we move forward, our script needs a means of error handling.

Enter Exceptions:

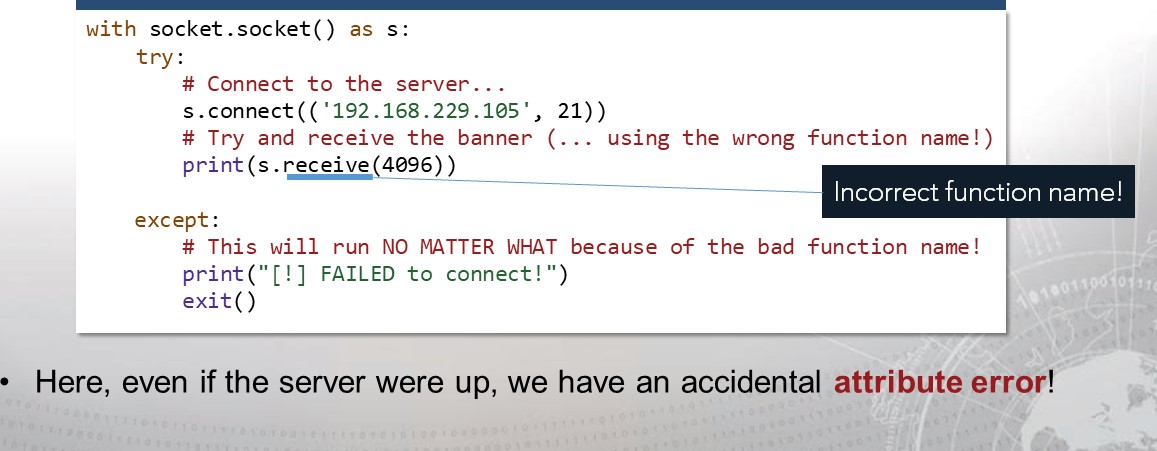
* Python error handling uses keywords try, except, raise, & finally
* You handle errors (or exceptions) by "try-ing" to do something:



* And that could go smoothly, "except" when something goes wrong!

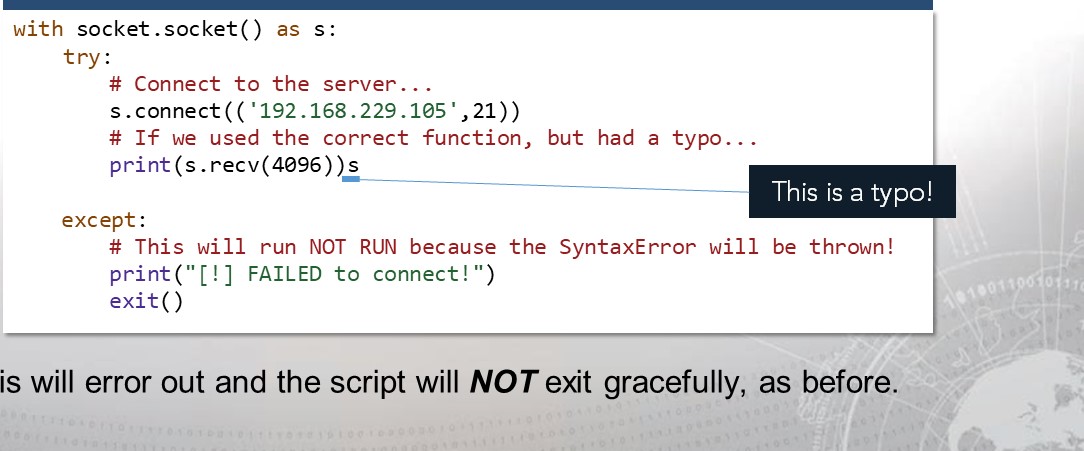
Be warned!

* Using a general except statement, any error that occurs will trigger it!



Syntax errors will not be caught.

* However, even with a general except statement, syntax errors will show.

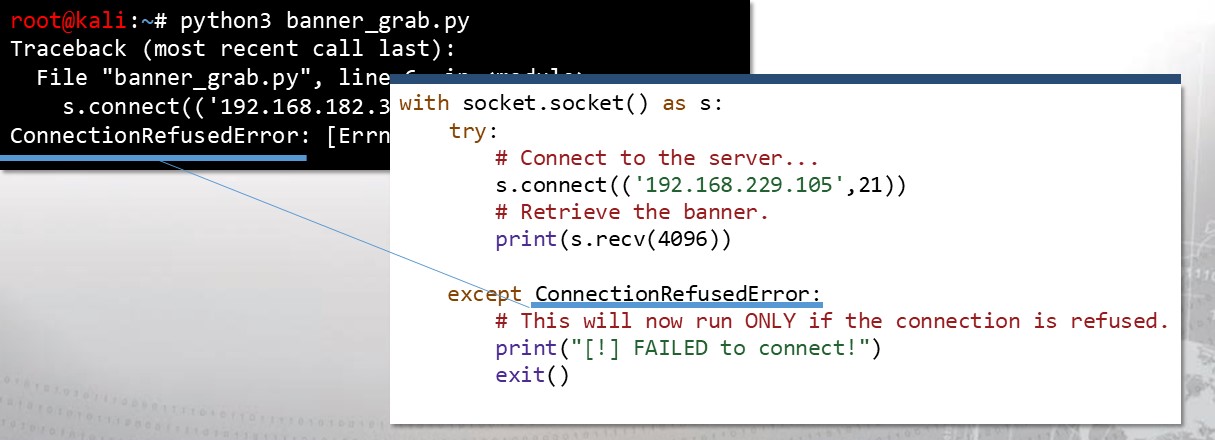


•

This

Be specific with your exceptions!

* To avoid catching any kind of error, provide a specific error type.



But there are still other errors...

* What if you need to handle more than just one kind of error?

root@kali # python3 banner\_grab. py Traceback (most recent call last) :

File "banner\_grab. py", line 6, in <module>

s. '192.168.119.105', 21))

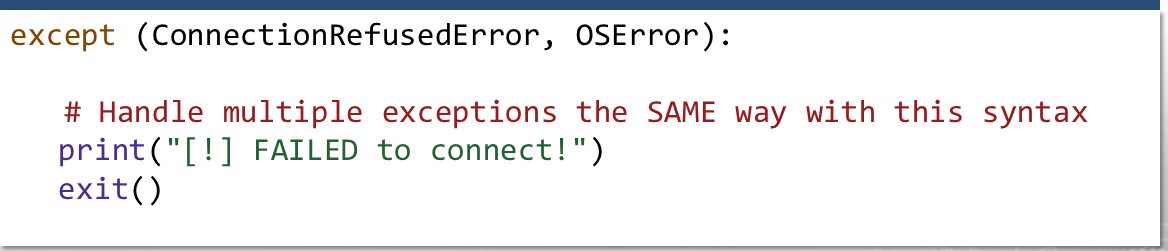
OSError: [Errno 113] No route to host

* In this case, perhaps the server cannot be reached because it is not at all within its range...



You can handle multiple exceptions easily!

* To handle these errors the same way, use a tuple with except.



* The only remaining case is handling different exceptions in different ways.



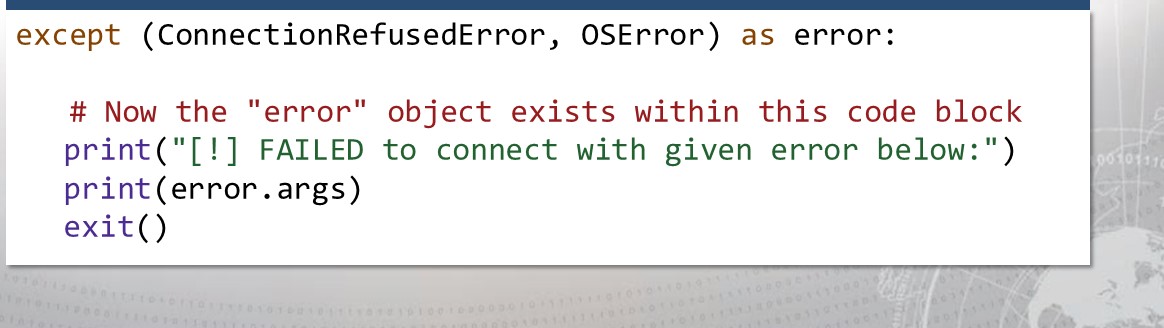
You can be distinct in how multiple errors are handled.

|  |
| --- |
| except ConnectionRefusedError:  # Be precise in error handling with specific separations of exceptions. print("[!] FAILED to connect, the server is refusing connections! ") exit()  except OSError:  # If you wanted to do something different in this case, you could ! print("[!] FAILED to connect, no route to host! ") exit() |

* To handle multiple errors differently, just add more except code blocks.

And you can generalize these caught exceptions.

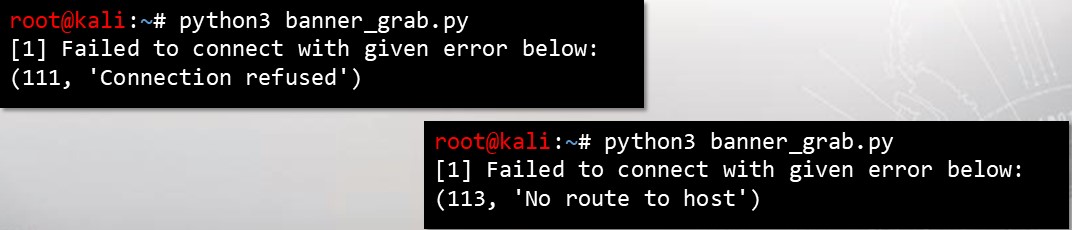
* Any except statement can keep track of the exception object it caught.
* So if you want to see the real error message that Python would originally give you, but still gracefully catch the error, you can use this syntax:



Better visibility on errors and cleaner code.

* The error object (or whatever you decide to call it) inherits properties from, at minimum, the Python BaseException.
* That allows you to see the args property, which is a tuple, like so:

[1] Failed to connect with given error below:



•

args[l]

=

error

message

•

args[ø]

=

error

number

(111, 'Connection refused')

The other keywords for exception handling: finally

|  |
| --- |
| with socket . socket() as s: try :  # Connect to the server...    except (ConnectionRefusedError, OSError) as error:  # There was an error! Tell the user.  FAILED to connect with error below: " print (error .args)  finally:  # Regardless what happens, do this "on the way out " print("The program will continue from here! |

* The finally statement will run after a try/except segment, regardless of whether or not an exception has been handled.

The other keywords for exception handling: raise

* The raise statement will force a specified error to occur.

|  |
| --- |
| >>> raise ValueError("Your custom error message! ")  Traceback (most recent call last):  File  , line 1, in <module> raise ValueError("Your custom error message! ValueError: Your custom error message! |

* This is most commonly used when you are writing your own module or classes and are preparing for potential errors that other programmers might run into.



Exception Handling Best Practices:

Minimize your try blocks.

Specify what you are wanting to catch with your except blocks

Generalize multiple errors by keeping



track of the Exception objects.



Test your inputs to see what other



exceptions your code should handle.



Documentation on Exception Handling:

* For more detailed functionality and syntax examples, view the Python tutorial on Errors and Exceptions: https://docs.python.org/3/tutorial/errors.html
* For other use cases and specific types of exceptions, view the Python documentation on Built-in Exceptions:

https://docs.python.org/3/library/exceptions.html

socket module reminders:

* Handle socket objects "with" a context manager.
* Input and output is simply send() and recv()
* Data is transferred in Python bytes objects.

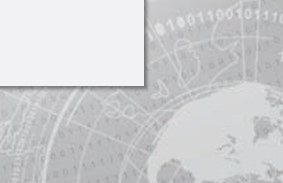


The urilib Module:

* You have automated the process of working with a network socket.
* This could be used to connect to port 80.
* But Python can do better: one of the standard libraries is urllib.

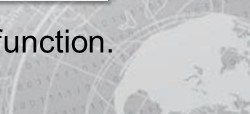


|  |
| --- |
| urllib — URL handling modules  Source code: Lib/urllib/  urllib is a package that collects several modules for working with URLs:  urllib.request for opening and reading URLs urllib.error containing the exceptions raised by urllib. request urllib.parse for parsing URLs urllib.robotparser for parsing robots . txt files |

urllib: Interacting with the Internet

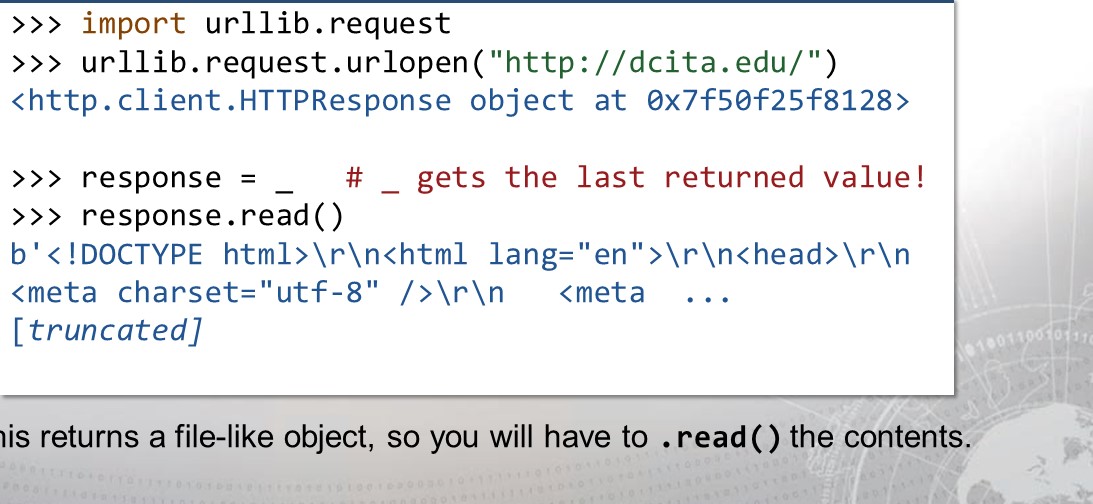
# Reading web pages with urllib . request

|  |  |  |
| --- | --- | --- |
| |  | | --- | | The urllib. request module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.  urllib . request . urlopen(url, data=None, [timeout, ]\*, cafile=None, capath=None, cadefault=False, context=None)  Open the URL url, which can be either a string or a Request object.  Source: https://docs.python.org/3. l/library/urllib.request.html | |  |

* Open web pages by using the urllib. request .urlopen() function.

Typically you supply a URL as a string:





•

This

You can use a context manager!

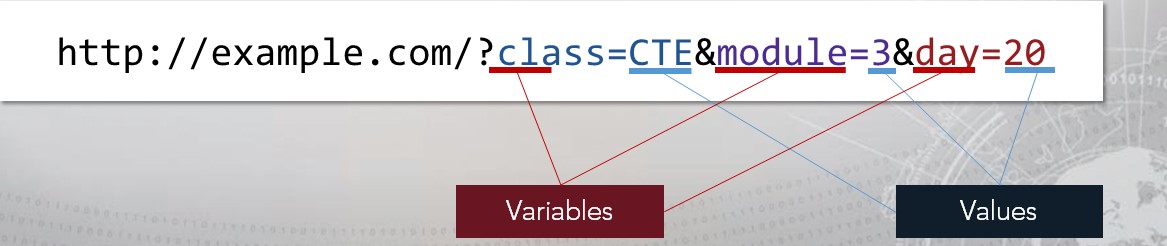


|  |
| --- |
| import urllib.request with urllib.request.urlopen("http://dcita.edu/") as response:  print(response. getcode(), response.geturl())    print(response. info())  # show headers  200 http://www.dcita.edu/  Content-Type: text/ html  Content-Length: 34356  Server: AmazonS3  Connection: close |

* The response object has plenty of other properties... see documentation!

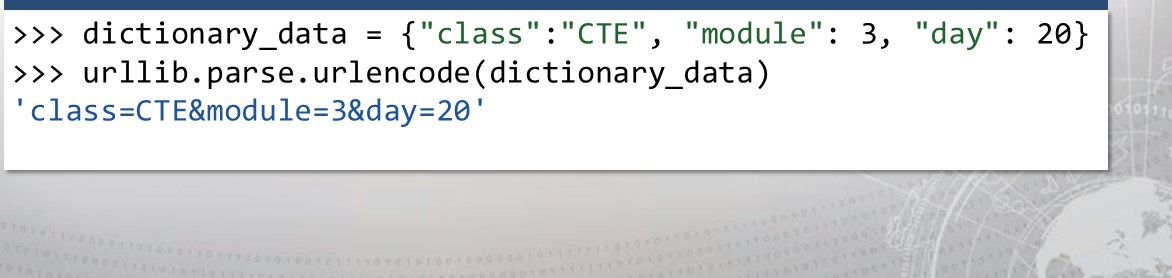
By default, you are sending HTTP GET requests.

* GET is the most common HTTP method, used for retrieving data.
* To send variables and data in the request with a GET method, you supply them as part of the URL, denoted by a question mark.
* Data is supplied in the form variable-value, joined by an ampersand.



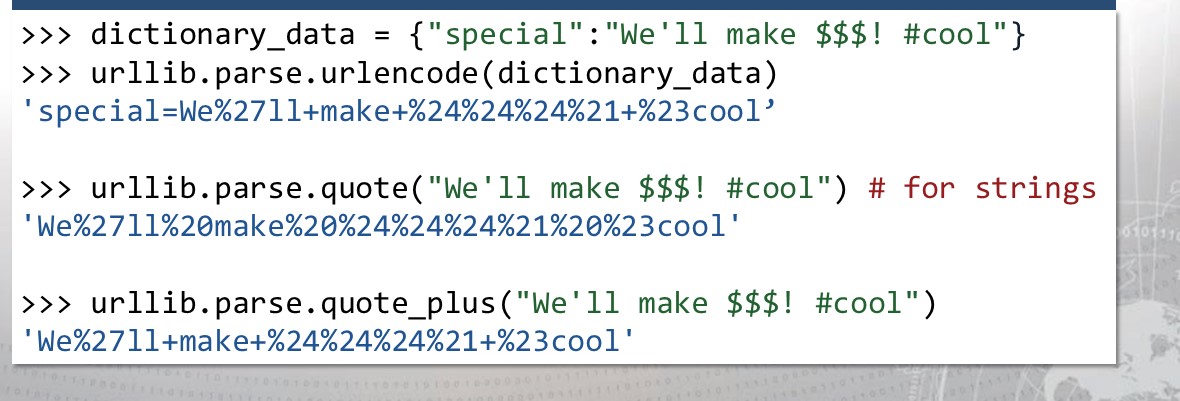
urllib offers functionality to easily put data in that form.

* The urllib submodule, urllib.parse offers a convenient function.
* urllib.parse.urlencode() takes a dictionary as an argument, and will convert it into the HTTP variable form.



And urlencode( ) will encode special characters.

* Appropriately given the name, the urlencode() function will also properly handle special characters passed into a URL.



You need to parse your data to make a POST request.

* To submit data (like filling out a form), you usually make a POST request.
* This is where the parsing functions come in handy.
* The data is passed as an argument must be encoded (as bytes!)

|  |
| --- |
| >>> post\_data 'class" . "CTE , "module" : 3, "day" 20}  >>> post\_data = urllib.parse.urlencode(post\_data)  >>> data\_bytes = post\_data.encode("ascii") # turn to bytes !  >>> urllib. request.urlopen("http://example.com", data\_bytes) |
|  |

I

urllib Resources and Reading Material

There is much more that the urllib module can do. We only touched upon the basics.

For more details, examples, and use cases, look at the official documentation.

https://docs.pvthon.orq/3/librarv/ur//ib.request.html#m odule-ur//ib. request

You will have plenty of opportunity to work with urllib in the exercise.

To avoid a lot of the overhead...

* The urllib module needed prep-work to be done for making requests.
* A great alternative, that is now seamlessly available to Python 3, is the requests module.
* requests turns HTTP methods into their own Python methods:

## • HTTP GET - requests.get("http://example.com"

* HTTP POST - requests . post( "http: //example. com" and so on



I

requests returns response objects in a simpler way.

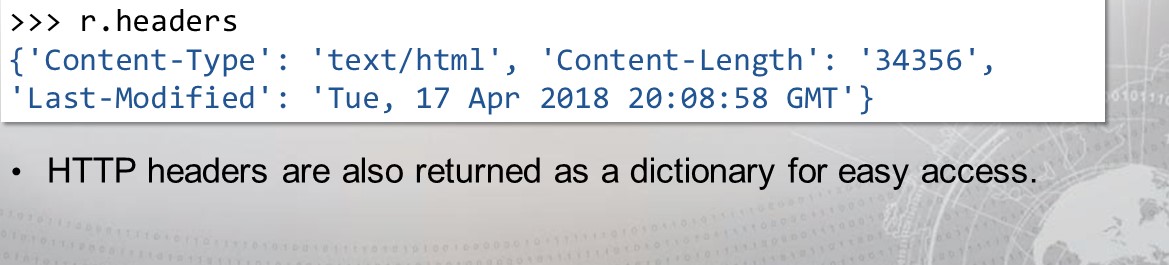


* The requests module typically makes for much less code. 



Supplying and accessing data is much faster.

|  |
| --- |
| >>> get\_data = {"class" : "CTE "module" : 3 "day": 20}  requests . get ( "http : //dcita.edu" , params = get\_data)  >>> r. headers  Content-Type' : 'text/ html ' , ' Content- Length ' : ' 34356 ' ,  ' Last-Modified ' : 'Tue, 17 Apr 2018  GMT' } |
|  |

* If you didn't want to bother putting GET parameters in the right form, the requests module can handle it passed as just a dictionary.

This is just as easy with POST data.

* You can do the same thing with an HTTP POST method.

|  |  |
| --- | --- |
| >>>  >>>  >>>  405  >>> | post\_data = {"class " . • "CTE" , "module" . 3, "day" : 20} requests . post( "http://dcita . edu", data = post\_data)  r. status code  # This is "Method Not Allowed" just in this example. |

* You should only POST data to pages supporting that method.

The requests module can do much more.

* File upload:

requests. post(url, files = {"filename" : open( "filename"

* Decode JSON data:

# requests . get(url); print(r.json())

* Handle timeouts:

requests.get(url, timeout

* Send custom headers or cookies:

requests.get(url, headers = h dict, cookies = c\_dict)

. it can do much, much more:

* Basic HTTP authentiction:

requests. get(url, auth ("username", "password"))

* Different HTTP methods:

requests. put(url); requests . patch(url); requests. head(url)

* Monitor redirections:

requests.get(url); print(r.history)

* Handle sessions and cookies:

requests . Session(); s. get(url); print (s. cookies)

Cookies can be stored as part of a Session:

* HTTP cookies can be passed along with a request (last slide). ..
* Or they can be modified relative to the "Session" they belong to.



I

# requests Resources and Reading Material

The requests module has a very simple syntax and a lot of functionality.

For more details, examples, and use cases, look at the official documentation.

http://docs.python-requests.org/en/master/ http://docs.pythonrequests.org/en/master/user/quickstart/

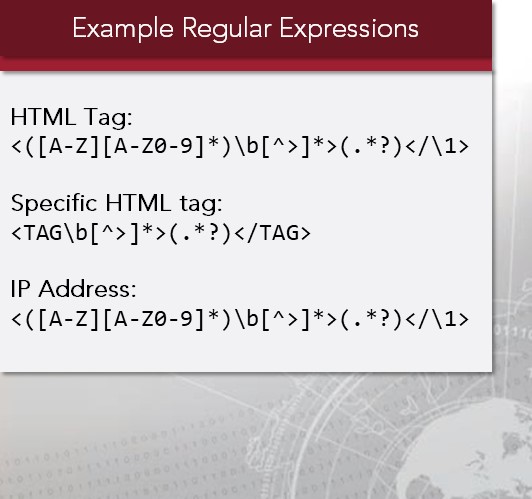
You will have plenty of opportunity to work with requests in the exercise.

So how do you process data you might get from a site?

* If you are have a very large string, you likely want to carve things out of it.
* You can muddle around with the string. split() syntax and slicing...
* But this is often inefficient, and Python can do better.
* Thankfully, there are modules to help with text processing!



Have you heard of regular expressions?

* Regular Expressions, or "regek' are strings of text that define a pattern that is used by algorithms to search for text, often used for "find & replace" or input validation.
* Typically, they look like gibberish.
* Each character has a special meaning.

Python has a built-in re module to work with these.



|  |
| --- |
| >>> import re  >>> pattern = re . compile(  >>> m = pattern. search("my example string for CT E")  <\_sre.SRE\_Match object; span=(22, 25), match= 'CTE' >  >>> m. group()  'CTE' |

* There are a lot of ways to find a match with re. This is only one example.

Regular Expression Crash Course (1/3):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Character Meaning Ex. Pattern Ex. Match | | | | |
| \w | "Word character" (letters, digits, underscores) \w\w\w\w NOT "word character' | cT3 |
| \d | Digits (0-9) version \d.\d  NOT digits | version 2.0 |
| \s | " Space characters " (tabs, newlines, vertical tab) a\sb\sc | a b c |
|  | NOT space characters | DC3CTA |

Any character

* As a rule, Regex patterns look at each character literally.

 With the EXCEPTION of the special characters defined in these tables.

Regular Expression Crash Course (2/3):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quantifier Meaning Ex. Pattern Ex. Match | | | | | |
|  | One or more repeats of the previous character |  | long\_w0rds |
| {3} | Three repeats of the previous character |  | 1337 |
| {2,4} | Two to four repeats of the previous character |  | AA or AAA |

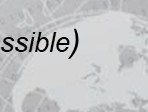
At least three repeats of the previous character

Zero or more repeats of the previous character

The previous character once or more (optional) plurals? plural

Makes quantifiers "lazy" (as little as possible) hello{3,8}? hellooo



* These quantifiers are, by default, "greedy" (match as much as possible)
* One of the most powerful regex is: .+ (match any character as much as possible)

Regular Expression Crash Course (3/3):

|  |  |
| --- | --- |
| Anchor | Meaning Ex. Pattern Ex. Match |

 Positioned at the start of the string/line. line start

Positioned at the end of the string/line. . \*end$ line end

Grouping, one of the characters in the braces  Dog or Dug

One of the characters NOT in the braces group  Dig

Captured grouping, a substring to extract <b>(.\*)</b> bolded text

OR operator in captured groups (this that) that

Contents of captured group #1  DC3CTA

* Captured groups let you select a portion of your pattern match.
* All these special characters and control make regex very powerful. 

The re module breaks down into two concepts:

* Python uses two high-level objects to handle regular expressions:

|  |  |  |
| --- | --- | --- |
| Regex Objects |  | Match Objects |
| Considered "compiled" patterns, that offer functions to perform operations like search, split and substitute on given text.  regex = re. compile("<b>(. | Returned from function calls on regex objects, with properties regarding the matched text like start and end positions.  match regex. match( "  " ) |



* The module also offers convenience functions that do the same the operations as Regex objects, but without "compiling" a pattern. 

"

Difference in "search ( )" versus "match ( ) 

* It is important to know the difference between the search() and match() operations, because you might accidentally trip up:

|  |  |  |
| --- | --- | --- |
|  |  | match() |
| search() will look for the first location that matches the given pattern. | match() will look to see if the beginning of the string matches the given pattern |

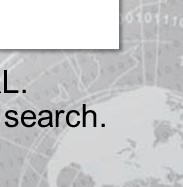
* More often than not, you likely want to use the search() function!



Greedy matching versus lazy matching:



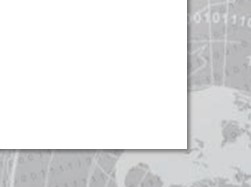
|  |
| --- |
| href="http://dcita.edu/">But "where do we learn, " you ask?</a>'  regex re. (. # Note this is GREEDY by default! print (regex . search(s) . group(l))  # http://dcita.edu/">But "where do we learn,  regex re. (. # LAZY matching gets what we want! print (regex . search(s) . group(l)) # http://dcita.edu/ |

* Say we had an HTML anchor tag and we wanted to extract the URL.

There is the potential to match too much using the default greedy search.

Often times you will want more than just the first match.

|  |
| --- |
| >>> import requests requests . get( " http : //dcita . edu" ) import re re.compile( '<a href="(http.\*?)" '  p. findall(r.text)  https://learn .dcita .edu',  ' http://www. council.org/',  ' http://www.airforce . com/' ,  ' http://www.afreserve.com/ ' ,  ' http://www.goang.com/',  ' http://www.dc3.mi1/',  ' http://www. usa.gov/',  ' https://dod.usajobs . gov/' ,  ' https : / /www.usajobs . gov/ ' |

To retrieve more than just one result, use

methods like findall() or finditer()

These will return only strings representing

the match (not a Match object!) packaged

inside of a list.

You can also supply "flags" to tweak even more settings...

As an optional keyword argument to most every Regex operation method, you can use flags (constants in the re module) to change the pattern:



* make \w, \d, \s and their variants match only ASCII.



* Perform case-insensitive matching.



* ensure characters like A and $ match line anchors.



* force the . to match all characters (including newlines)

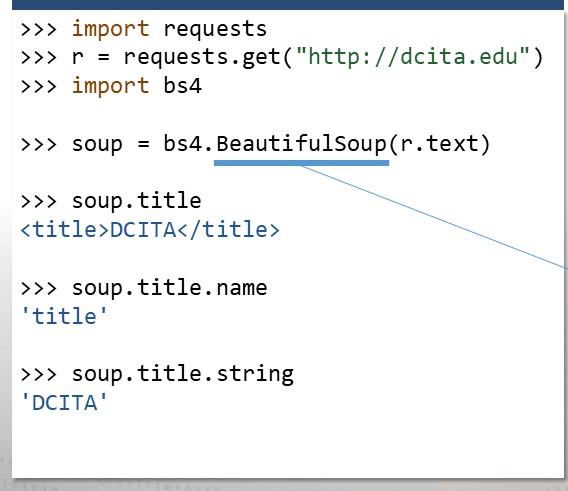
# re Resources and Reading Material

* Regular Expressions are extremely versatile and they are used in so many other applications and programming languages!
* To practice and experiment with more regex, check out: https://regexr.com/ or https://gchq.github.io/CyberChef/
* For more details, examples, and use cases for the re module, look at the official documentation: https://docs.python.org/3/library/re.html

Use BeautifulSoup for web scraping:

* The purpose of text processing so far has been strictly web processing... but Regular Expressions are general-purpose and can do so much more!
* A more tailored library specifically to do web scraping is bs4.
* bs4 is accessible in Python3. It will take an HTML document and turn it into a tree of Pythonic objects that you can navigate through and manipulate.

BeautifulSoup lets you extract data through objects.

The module will parse through HTML

and offer access to each element and attribute.

The most high-level object is the

'BeautifulSoup", and you can drill

down from there.

It breaks down into four conceptual objects:

The module uses...

* 1. A Beautiful Soup object, as the

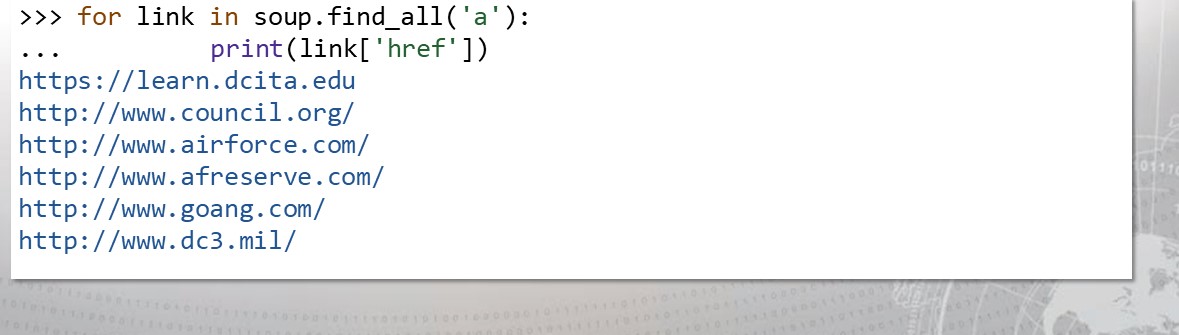
top-level tree

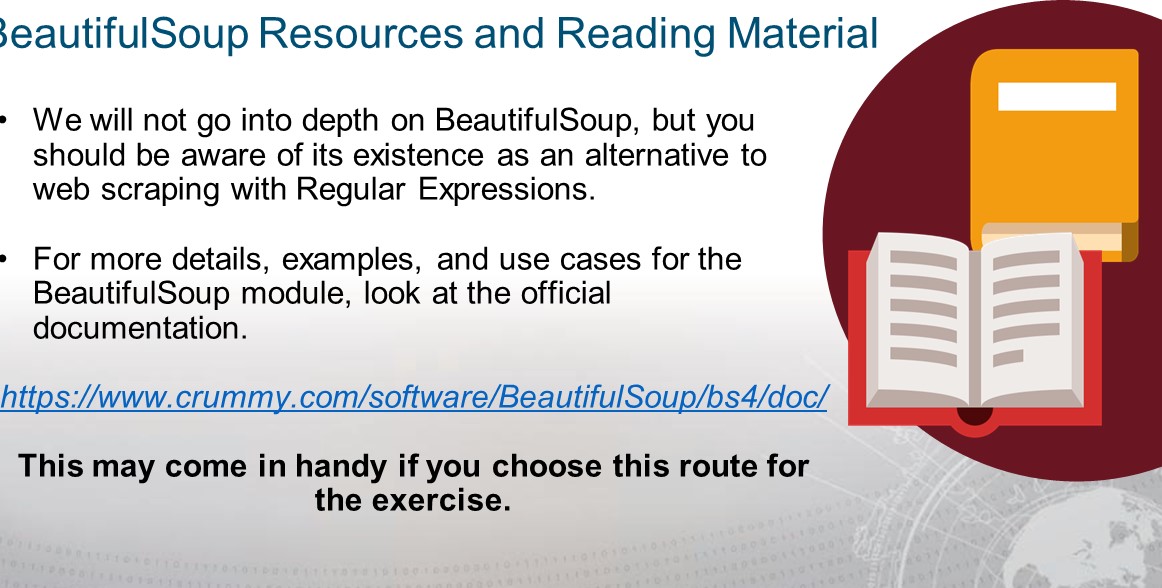
* 1. A Tag object, as an HTML tag in the original document
  2. A NavigableString, as a bit of text within a tag.
  3. A Comment, as a special errata of a NavigableString. 

BeautifulSoup is handy for finding multiple elements:

* You can access element (Tag) attributes by treating it like a dictionary.
* This example finds links just as we did with Regex, but is more readable:





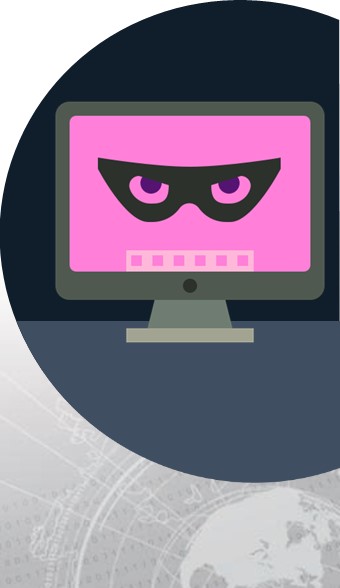


BeautifulSoup

•

•

Often times, cyber threats mask themselves:

* When an attacker or an adversary wants to hide their payload, they will obfuscate their code, or the data that they are working with.
* This can be done in many ways, and some methods offer a stronger means of "protection" in how sophisticated the obfuscation is.
* Typically this is done to avoid signature detection, or even just to add layers of complexity so defenders are less likely to find the real payload.

One very common method is simple data encoding.

* While it is a very weak form of obfuscation, it is certainly the most common: just encoding data into another form or representation.
* This is trivial because the only thing necessary to de-obfuscate is to decode the encoded data. Surprisingly, this is extremely prevalent.



A very predictable method is Base64.

* Base64 is a binary-to-text encoding scheme that represents data in an ASCII string, using only printable characters, like letters and numbers.
* It is a form of encoding. For every one string of data decoded, there exists



only

one

string

encoded

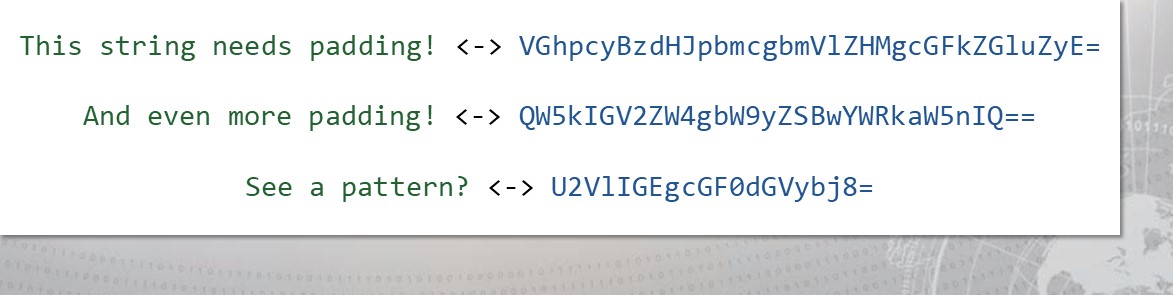
and

vice

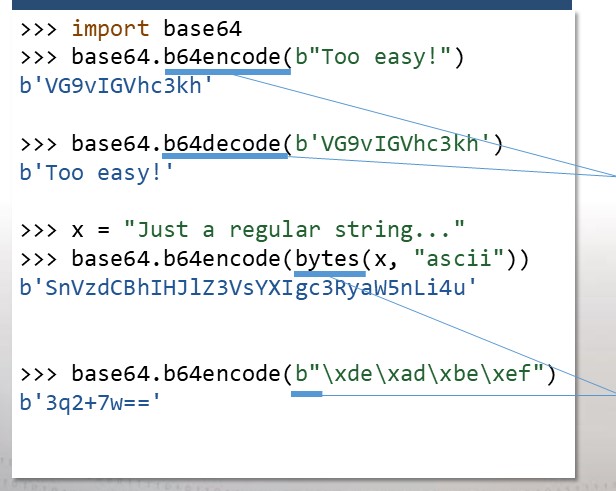
versa.

Base64 is very recognizable:

* As a rule, Base64 encoding must have a length as a multiple of four.
* If an encoding does not have a length as a multiple of four, it adds up to two trailing equals signs (z) as padding.



Python has a built-in library for it:

The base64 module in Python has two

simple functions:

b64encode() and b64decode()

Python 3 requires the arguments be

passed as bytes, so you can prepend

your string with a "b" or use the bytes() function.

Base64 is just a number base, like any other:

* You know base 10 (decimal), base 2 (binary), and base 16 (hexadecimal).
* They are all just another way to represent the same data. Just as there is Base64, you could also find Base32 or even Base85/Ascii85:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Base32 |  | Base85/Ascii85 | | Uses only uppercase letters and the numbers 2-7. Pads with equal signs to a length as a multiple 8.  base64.  ) | Uses  letters,  numbers,  and  punctuation | |  |

# base64 module Resources and Reading Material

* Base64 and its variants are not conceptually hard to grasp...
* The priority is instead learning to recognize and identify it when you see it.

For more details, examples, and use cases for the base64 module, look at the official documentation.

https://docs.python.org/3.4/1ibrary/base64.html

You will have plenty of opportunity to work with base64 in the exercise.

I

Exercise: Python Modules

Objectives

After completing this exercise, students will be able to:

* Conduct active reconnaissance using Python
* Plan exploitation using Python
* Interpret Python module documentation
* Manipulate web content with Python

Duration

This exercise will take approximately 4 hours to complete, with 30-45 minutes to review answers.

I

Debrief

General Questions

* How did you feel about this procedure?
* Were there any areas in particular where you had difficulty?
* Do you understand how this relates to the work you will be doing?

I

Debrief

Specific Questions

* In step 5 you found a Local File Inclusion vulnerability. What other potential files could you retrieve?
* Could you leverage the Local File Inclusion vulnerability to a much more severe vulnerability? If so, how?
* Why did you need to maintain a "session" for steps 7 and onward?
* In step 14 you found a file upload feature. How could this be abused to open more vulnerabilities? How and why?

I

# Lesson Summary

In this lesson we discussed:

* Exception Handling
* Standard Libraries Tour
* urllib / requests
* re / BeautifulSoup

. base64

I

|  |
| --- |
| End of Module 2, Less o n  1 1 |