* Course Overview
* Module Overview
* Prerequisites and Course Outline
* HTTP Requests and Responses
  + URL: resource locator
    - Allows you to uniquely identify where you want to get your content from
  + Browser makes http request to server to where resource is located
  + Server return http response
  + HTTP: hypertext transfer protocol
    - Simple ubiquitous text-based protocol used by browsers and apps to access web content
  + HTTP Clients
    - Client server protocol
    - Web browsers
    - Mobile apps
    - Programs
  + HTTP Servers
    - Host web pages and web content
    - Static or dynamic(javascript add interactivity to web page)
  + HTTP Requests
    - Clients make http requests
    - GET: fetch resources
    - POST: create/update resources
    - PUT: idempotently create/update
    - HEAD: to get only http header
    - DELETE: to delete resources
  + HTTP Responses
    - Servers are standing by to field requests
    - Send back http responses
      * Status line with code such as 404, 200, etc
      * Response header with metadata
      * Response body
* Web Scraping
  + Automated extraction of data from websites
  + Website content is first fetched(usually using HTTP) and then parsed to extract specific information
  + Web Pages
    - Websites are collections of web pages
    - Web pages consist of markup
    - This markup is understood and rendered by browsers
  + Fetching and Parsing
    - The same HTML markup can be accessed(fetched) via HTTP
    - Possesses an in-built hierarchical structure
    - Parsers can exploit this structure to extract information
* HTTP Client Libraries
  + Web scraping
    - Fetching content through http request through client library
    - Parsing content(HTML parsing, DOM Parsing, Computer Vision)
  + Fetching content
    - Urllib, Urlib2, Requests, Httplib, Httplib2
  + Requests
    - High level api
    - Easy to use
  + Httplib2
    - Fine-grained control of http request
  + Urllib
    - Part of python standard library
    - No need to download
  + Web servers make content available on HTTP endpoints
  + Browsers make HTTP requests under-the-hood to get web pages
  + Web scraping usually involves making such requests programmatically
  + Command-line HTTP requests
    - Curl
  + Python libraries for programmatic access
    - Requests
    - Httplib2
    - Urllib
* Making GET Requests Using h.
  + Httplib is an http client library which allows fine-grained access to HTTP requests
  + Httplib2 is goolge’s more powerful implementation
  + In ‘New’ Select python 3 as kernel
    - in the ‘In []’ slot type :!pip install httplib2
    - click run
  + in another ‘In []’
    - import httplib2
    - import webbrowser
    - from pprint import pprint
    - click run
  + in another
    - type url
    - ex) bin\_url = ‘https://httpbin.org/’
  + in another
    - invoke the web browser
    - type ‘ webbrowser.open(bin\_url)
    - will open page in new tab
  + in another
    - instantiate the http object to make http requests
    - type ‘http = httplib2.Http()
  + in another make a http request
    - resp, data = http.request(bin\_url)
    - returns response header(resp) and response body(data)
  + type(resp)
    - returns type of resp
  + len(resp)
    - returns length or response
  + resp.status: return status code
  + resp.reason: returns status code message
  + resp.version: returns version
  + resp.previous: contains original url incase url was redirected
    - if return empty, then there was no redirect
  + data
    - will contain the html markup
    - will be type of bytes
  + html = data.decode(“UTF-8”)
    - convert html to string format
  + http.request(url)
    - makes a get request to the url
  + different urls will have their data encoding in different formats
  + ex) data.decode(‘ISO-8859-1’), data.decode(‘UTF-8’)
* Making OPTIONS, POST, PUT ..
  + Can make different http requests to perform different types of actions
  + To know what type of requests is support at url, make http request with method = ‘OPTIONS’, then check the header
    - resp, data = http.request(bin\_url, method = ‘OPTIONS’)
    - pprint(resp)
  + method = ‘HEAD’ to only get the head and not the body
    - resp, data = http.request(bin\_url, method = ‘HEAD’)
  + when you don’t specify a method, it defaults to ‘GET’
  + POST: send data to server
    - post\_data = ‘{“name”: “Alice”, “college”: “Harvard”}’
    - resp, data = http.request(‘https://httpbin.org/post’, method = ‘POST’, body = post\_data, headers = {‘content-type’:’application/json’})
  + PUT: similar to post but idempotent
    - Can make the same put request multiple times and it will return the same result
* Handling Redirects with httpli..
  + Follow only safe redirects where a GET or a HEAD request is made
    - http.follow\_redirects
  + follow redirects for all kinds of requests
    - http.follow\_all\_redirects
  + resp.previous is not null if redirect happened
* Making HTTP Requests and P…
  + Urlib, urlib2 are high level interface for fetching data from URLS
  + Ex
    - import urllib
    - import webbrowser
    - from pprint import pprint
    - bin\_url = ‘https://httpbin.org/’
    - webbrowser.open(bin\_url)
  + urllib.request.urlopen(bin\_url)
    - resp = urllib.request.urlopen(bin\_url)
    - download contents of url
  + resp.url(): gets url
  + resp.getcode(): gets status code
  + resp.info(): gets the header of the response
  + resp.read(): to get content that have been downloaded
    - acts like a file handle
    - after reading the first time, it will return empty
  + use scraping tools such as beautiful soup to parse the response
  + resp.reason: to get status message
  + to make other type of http requests you need to instantiate a urllib request object
    - req = urllib.request.Request(bin\_url, method = ‘OPTIONS’
    - resp = urllib.request.urlopen(req)
  + to make a post request, you need to encode post data
  + ex)
    - post\_data = urllib.parse.urllencode({“name”: “Alice”, “college”: “Harvard”}).encode(‘ascii’)
    - req = urllib.request.Request(‘https://httpbin.org/post’, method = ‘POST’, data = post\_data, headers = {‘User-Agent’: ‘Mozilla/5.0 (Windows NT 6.1; x64)’})
    - resp = urllib.requst.urlopen(req)
  + good practice to make request with ‘with’ blocks
  + ex)
    - with urllib.request.urlopen(‘https://www.google.com/search?q=pluralsight’) as resp:
    - pprint(resp.read().decode(‘ISO-8859-1’))
  + With in python is used when working with unmanaged resources(like file streams)
    - It is similar to ‘using’ statement in C#
  + To handles errors
    - from urllib.error import URLError, HTTPError
    - try: …
    - except HTTPError as e: …
  + URLError is a more generic error
  + urllib.parse.urlparse
    - has specific functions that allows you to parse url
  + every url is made up of sub components which can be parsed
    - scheme, location, path
  + parsed\_data.scheme
  + parsed\_data.netloc
  + parsed.query
  + parsed.fragement
    - frag components is everything after #
    - often used as a bookmark in the page
* GET and POST Requests Using…
  + Requests is an easy and intuitive API to work with HTTP request and responses
  + Install library
    - !pip install requests --upgrade
  + Set up
    - import requests
    - import json
    - import webbrowser
    - from pprint import pprint
  + get request
    - resp = requests.get(‘url’)
  + resp.status\_code returns the status code
  + resp.headers to get the header
  + resp.text to get actual content
  + json.loads(resp.text) to parse to json
    - you can then look up values using keys
  + example using wikipedia
    - wikipedia\_search\_url = ‘https://en.wikipedia.org/wiki/Special:Search’
    - webbrowser.open(wikipedia\_search\_url)
    - query\_params = {‘search’: ‘requests’}
    - resp = requests.get(url = wikipedia\_search\_url, params = query\_params)
    - will redirect us to the first found result
  + requests.post to post data
    - post\_data = { ‘title’: ‘some\_title’, ‘body’: ‘some\_body’, ‘userId: 22 }
    - resp = requests.post(‘https://jsonplaceholder.typicode.com/posts’, data=post\_data)
  + requests library you won’t have to encode data
* Handling Redirects with the R..
  + Can force not to follow redirects
  + Ex)
    - resp = requests.get(‘url’, allow\_redirects=False)
  + can also specify timeout
    - requests.get(‘http://google.com, timeout=0.001)
    - specify timeout=None if you want to wait until it is resolved
* Module Summary
* Module Overview
  + Use beautiful soup to parse html
  + The HTML tree structure
* The HTML Parse Tree
  + Html markup is used to define the structure and content on web page
  + HTML: Hypertext Markup Language is the standard markup language for documents designed to be displayed in a web browser
* Introducing Beautiful Soup
  + Python library for extracting data from XML and HTML files, creates a parse tree and offers intuitive ways to navigate the tree
  + Set up environment
    - !pip install BeautifulSoup4
    - !pip install lxml
    - !pip install html5lib
  + Set up project
    - import requests
    - import webbrowser
    - form bs4 import BeautifulSoup
  + soup = BeautifulSoup(very\_simple\_html)
  + soup.prettify()
    - pretty print the html content
  + soup.title
    - to print the title
  + soup.[tag] to print the tag
  + soup.[tag].[name]
    - print tag name
  + soup.[tag].[parent]
    - returns the parent of the tag
  + if there are multiple tags, soup.[tag] will return the first tag found
  + default parser is lxml parser
  + can specify the parser when creating the BeautifulSoup object
    - soup = BeautifulSoup(html, ‘lxml’)
  + html5 parses pages as a web browser does, creates valid HTML5
    - soup = BeautifulSoup(html, ‘html5lib’)
* Extracting Specific Page Elem…
  + soup = BeautifulSoup(resp.text, ‘lxml’)
  + Opening a file in your directory
    - with open(‘scraping-first-web-page-python/files/GiantPanda.html’) as html\_cdoe:
    - soup = BeautifulSoup(html\_code, ‘lxml’)
    - print(soup.prettify())
  + Getting specific elements
    - tag = soup.title
  + Every html element present in web page is available as a tag in beautiful soup
  + Every tag extracted from soup has a name, the name is the name of the html element
  + Can update the name of the tag
    - tag = soup.h1
    - tag.name #outputs ‘h1’
    - tag.name = ‘user\_defined’
    - tag.name #outputs ‘user\_defined’
  + Can grab attributes of the tag
    - Access specific attributes of tag using []
    - tag = soup.a
    - tag[‘href’] #outputs the value of the href attribute
  + Can grab all attributes of the tag using .attrs
    - tag.attrs
    - Returns a dictionary of attributes
  + Can also get attributes using .get\_attribute\_list(‘attribute’) to get a list of attributes
    - tag.get\_attribute\_list(‘class’)
  + Can get content of element using .string
    - tag.string
  + Can access comments
    - comment = soup.i
    - Comment
* Filtering Elements Using Find..
  + Use find() and findAll() to search for and filter elements on an HTML page
  + Set up project
    - import re
    - from bs4 import BeautifulSoup
  + Can find particular elements using soupObject.tag or soupObject.find(‘[tag]’)
    - Return first instance of that element
  + To get all elements use findAll
    - soup.findAll(‘p’)
    - Returns a list of all p elements
  + Can also use find and findAll as filter for search terms
  + Can also use to search for elements using attributes values
    - soup.findAll(src = ‘TomAndJerry.jpg’)
    - soup.find(‘p’, attrs = {‘class’: ‘comedy story’})
    - soup.find(id = re.compile(‘2’))
  + Can also filter by regular expression using re.compile
    - soup.find(re.compile(“^a”))
    - for tag in soup.findAll(re.compile(“^a”)):
    - print(tag)
  + Can use to search for specific content
    - soup.find(‘a’, attrs = {‘class’: re.compile(‘^cre’)})
* Searching and Filtering Using ..
  + Can specify multiple tag to search for
    - soup.findAll([‘a’, ‘img’])
  + can also extract all elements
    - for tag in soup.findAll(True):
    - print(tag.name)
  + Can create functions to filter
    - def has\_src\_but\_no\_href(tag):
    - return tag.has\_attr(‘src’) and not tag.has\_attr(‘href’)
    - soup.findAll(has\_src\_but\_no\_href)
    - def has\_no\_tom(href):
    - return href and not re.compile(“Tom\_Cat”).search(href)
    - soup.findAll(href = has\_no\_tome)
    - def has\_wikipedia\_source(href):
    - return href and re.compile(“wikipedia”).search(href)
* Extracting Links from a Page
  + <https://mashable.com/2014/08/28/static-website-generators/>
  + Project setup
    - import requests
    - import re
    - import webbrowser
    - from bs4 import BeautifulSoup
  + request library
    - resp = requests.get(url)
  + using beautiful soup to parse html
    - soup = BeautifulSoup(resp.text, ‘lxml’)
  + using beautiful soup to find all links
    - soup.find\_all(“a”, attrs = {‘href’: re.compile(“^https)})
  + to print only the href of links
    - for link in soup.find\_all(“a”, attrs = {‘href’: re.compile(“^https)}):
    - print(link[‘href’])
  + Regular expression, matching http or https
    - for link in soup.findAll(‘a’, attrs = {‘href’: re.compile(“^http|https”)}):
    - print(link[‘href’])
    - for link in soup.findAll(‘a’, attrs = {‘href’: re.compile(“^http”)}):
    - print(link[‘href’])
  + Finding all a links with href values
    - for link in soup.findAll(‘a’, href=True):
    - print(link[‘href’])
  + Making all relative links absolute links
    - for link in soup.findAll(‘a’, href=True):
    - if not link[‘href’].startswith(‘http’):
    - link = url + link[‘href’].strip(‘/’)
    - else:
    - link = link[‘href’]
    - print(link)
* Using a Soup Strainer to Parse ..
  + SoupStrainer allows you to choose which parts of the document you want to parse, limit parsing to a subset of a document
  + Project set up
    - import re
    - from bs4 import SoupStrainer, BeautifulSoup
  + to only get div tags
    - div\_tags = SoupStrainer(“div”)
    - soup = BeautifulSoup(html\_code, ‘lxml’, parse\_only = div\_tags)
  + also works for attributes
    - alt\_attr = SoupStrainer(alt = ‘create\_image)
    - soup = BeautifulSoup(html\_code, ‘lxml’, parse\_only = alt\_attr)
  + also work with regular expressions
    - id\_attr = SoupStrainer(id = re.compile(“link”))
* Module Summary
* Module Overview
  + Scrapy shell
  + Scrapy as an application framework for crawling websites
  + Data extraction in a structured format
  + Interactively testing extraction using scrapy shell
  + Leveraging Xpath and CSS selectors in the scrapy shell
* Parsing Web Content
  + Web Scraping
    - Fetching Content
      * Http Requests
    - Parsing Content
      * HTML Parsing
      * DOM Parsing
  + HTML Parsing
    - Parse HTML and CSS associated with web content(static content)
    - Can not parse dynamic changes made by javascript
    - Work with structured text rather than object of the document object model
    - BeautifulSoup, RegEx, Grep
  + DOM Parsing
    - Parse dynamic content(js) in addition to static content
    - Can parse dynamic changes made by code elements
    - Can work either with text elements or with objects in the DOM
    - Scrapy, Selenium
  + Scrapy is a framework that combines all of this(HTTP requests, HTML Parsing, DOM Parsing)
    - Framework for building produciton-grade, heavy-duty web parsing systems
  + Computer vision
    - Sikuli, visual technology to automate and test GUI
* Introducing Scrapy
  + Application framework for crawling web sites and extracting structured data
  + Originally built for web scraping but now used for web crawling
  + Web Scraping
    - Extract data directly from web sites
    - Data analysis and somewhat unsavory reputation
    - Specific - “scrape prices from Amazon”
    - Small scale, results in specialized dataset
  + Web Crawling
    - Download and index web sites
    - Performed by search engines associated with legitimate use
    - General - “crawl sites linked of Amazon”
    - Large scale, results in document corpus
  + Framework vs library: inversion of control
  + Library
    - You call library functions
    - You write application and invoke library for specific portions
  + Framework
    - Framework calls you
    - Framework defines the application and invokes your code for specific protions
  + Beautiful soup is parsing library
  + Scrapy is a web scraping framework
  + Specific HTML elements are selected for processing using selectors
  + Selectors
    - Specification of what HTML elements ought to be selected for processing
    - Scrappy supports XPath and CSS selectors
* Getting Started with Scrapy
  + [**https://stackoverflow.com/questions/19687421/difference-between-beautifulsoup-and-scrapy-crawler**](https://stackoverflow.com/questions/19687421/difference-between-beautifulsoup-and-scrapy-crawler%3C/span%3E)
  + Set up environment
    - pip install scrapy --upgrade
  + in cmd
    - type ‘scrapy’
    - will bring up command page
  + scrapy version
    - shows current version of scrapy installed
  + scrapy bench
    - runs a quick bench mark on your local machine to check how scrapy performs on your hardware
  + scrapy fetch --nolong ‘url’ > ‘filename’
    - downloads the HTML file and writes its contents out to a file
    - static content only for default scrapy
    - need extensions for dynamic content
  + scrapy settings can be adjust to change how you scrape
  + scrapy view ‘url’
    - open the url in the browser the way scrapy will “see” it, dynamic content will not be rendered
    - will download the html content and save it on your machine
    - when you use scrapy shell to prototype, scrapy will work on that local file
* Introducing the Scrapy Shell
  + Interactive shell where you can prototype how you will crawl a site without building a Spider
    - scrapy shell ‘url’
  + Scrapy will then make a request the web page and download its html and instantiate a bunch of scrapy object to work with
  + shelp()
    - Bring up all the details about the available scrapy objects
  + spider
    - Contains definitions and details of how a site will be scraped and the information to be extracted
  + Can specify your own custom spider
  + scrapy
    - Object which is the main module that gives you access to other scrapy object like requests
  + crawler
    - API that provides access to Scrapy’s components such as the execution engine, settings, etc
  + item
    - Container for structure data that you extract from a web page
  + request
    - Get you access to the url request that was made to the web page
  + response
    - Response that scrapy receives from the url target
  + response.url
    - Shows you the content of the url that was loaded
  + response.status
    - Gives you the status code returned by server
  + view()
    - Opens the parameter in browser
    - Copy the url of the local file
  + exit() to exit
  + scrapy shell ‘local file’
    - To work on the local version
    - Good so that change doesn’t affect you
  + fetch(url)
    - Fetch contents of an new url and reloads scarpy shell with contents of new urls
* Selecting Elements Using CSS
  + Ex url: scrapy shell https://www.webnots.com/basics-of-a-static-website
  + Use chrome inspector to find elements CSS selector and XPath
  + response.css(‘title’)
    - Gives the selector object for the title html element
  + response.css(‘title’).get()
    - Get the underlying html element specified by the selector
    - Returns the content of the first element that was selected
  + response.css(‘title::text’). get()
    - To get the text of the html element
  + response.css(‘p::text’).getall()
    - To access contents of all elements
  + response.css(‘p::text’).extract()
    - Equivalent of get all
  + response.css(‘p::text’).extract\_first()
    - Extract just the first text content that matches the selector
  + response.css(‘img::attr(src)’).extract()
    - Extract all images and return the src attribute
  + response.css(‘img::attr(src)’).extract()[1]
    - Gets the src attribute at index 1
  + response.css(‘img’).attrib[‘src’]
    - Equivalent to extract()
* Advanced Selections Using C..
  + Need to understand structure of web page
  + Ex) response.css('#post-149 > div.entry-content.clearfix > ol > li::text').extract()
  + Ex) response.css(‘#like-149 > span::text’).get()
  + Scrapy has download content of original page so changes to live page will not be available in shell until you restart the shell
* Selecting Elements Using XPa..
  + resposne.xpath(‘/html/head/title/text()’).get()
    - Gets the text of the title path using xpaths
  + Xpath allows you to walk down the html hierarchy of your webpage
  + /: forward slash indicates the absolute path to the tag starting at the root of the document
  + //: select all descendents which match this path at or below the root of the document
  + get()
    - Gets the first instance of the tag
  + extract()
    - Gets all instances of the tag
  + extract().first()
    - Gets the first instance of the tag
  + response.xpath(‘//img/@src’).extract()
    - To get all attribute src of all instances of the img tag
  + //\*: select any node with this characteristics
    - \* is wild card selector
  + “//\*[contains(text(), ‘static’)]/text)
    - Extract all text in web page that contains the text static
  + If you use response.xpath(“”) with .extract() you will get the selectors
* Module Summary
  + Beautiful soup is used for web scraping
  + Scrapy is an application framework for crawling websites
* Module Overview
  + Define crawling tasks in classes using spiders
  + Use item loaders to automate data extraction
  + Item loaders: allows you to zoom into portion you are interested in an extract data
  + Chaining data transformation using data pipe lines
* How Scrapy Works
  + Scrapy Engine: Controls data flow and triggers all events
  + Spiders: classes written by the user
    - Define how to parse responses and extract items
  + Engine forwards request to the Scheduler
  + Engine also asks for the next Request from the Scheduler
  + Scheduler responds with next Request for Engine to process
  + Engine requests the Downloader to get this from the internet
  + The Downloader fetches the URL from the internet
  + The Downloader then sends it back to the Scrapy Engine
  + The Engine forwards that Response object back to the Spider
  + The Spider processes the Response and returns scraped items
  + The downloaded items are sent to Item Pipelines
  + Items Pipelines are used for cleaning, validation, persisting to database, etc
  + Spiders
    - Custom classes where you define custom behavior for crawling and parsing pages from a site or group of sites
  + Implementing Spiders
    - What to crawl: URLs to start with are in the start\_request() method
    - How to crawl: Callback function inputs web page and outputs Items, Requests etc
    - How to parse: selectors which determine which parts of web page are processed
* Creating Your First Custom Sp..
  + <https://finance.yahoo.com/sector/ms_technology>
  + Create a project
    - scrapy startproject SimpleSpider
  + cfg file
    - contains configuration setting for scrapy project
  + SimpleSpider folder
    - Contains starter files to plug into scrapy framework
  + Spiders folder
    - Actual code for the spider to parse website and extract information
  + genspider command
    - ex) scrapy genspider [spider\_name] [url]
    - create a new spider file manually or use genspider command
    - creates a file with template for a basic spider
  + inside of the file
    - have to import scrapy
    - class inherits from scrapy.Spider
    - name: used to invoke and execute spider
    - allowed\_domains: restricts what domain spider parses
    - parse method: where you specify how to want to parse the contents of web page
  + def parse(self, response):
    - company\_names\_list = resposne.xpath(‘//\*[@id=”src-res-table’]/div[1]/table/tbody/tr/td[2]/text()’).extract()
  + move to the outer director where you created the SimpleSpider project
    - to run the spider
    - ‘scrapy crawl [spider\_name]’
* Writing Scraped Contents to a …