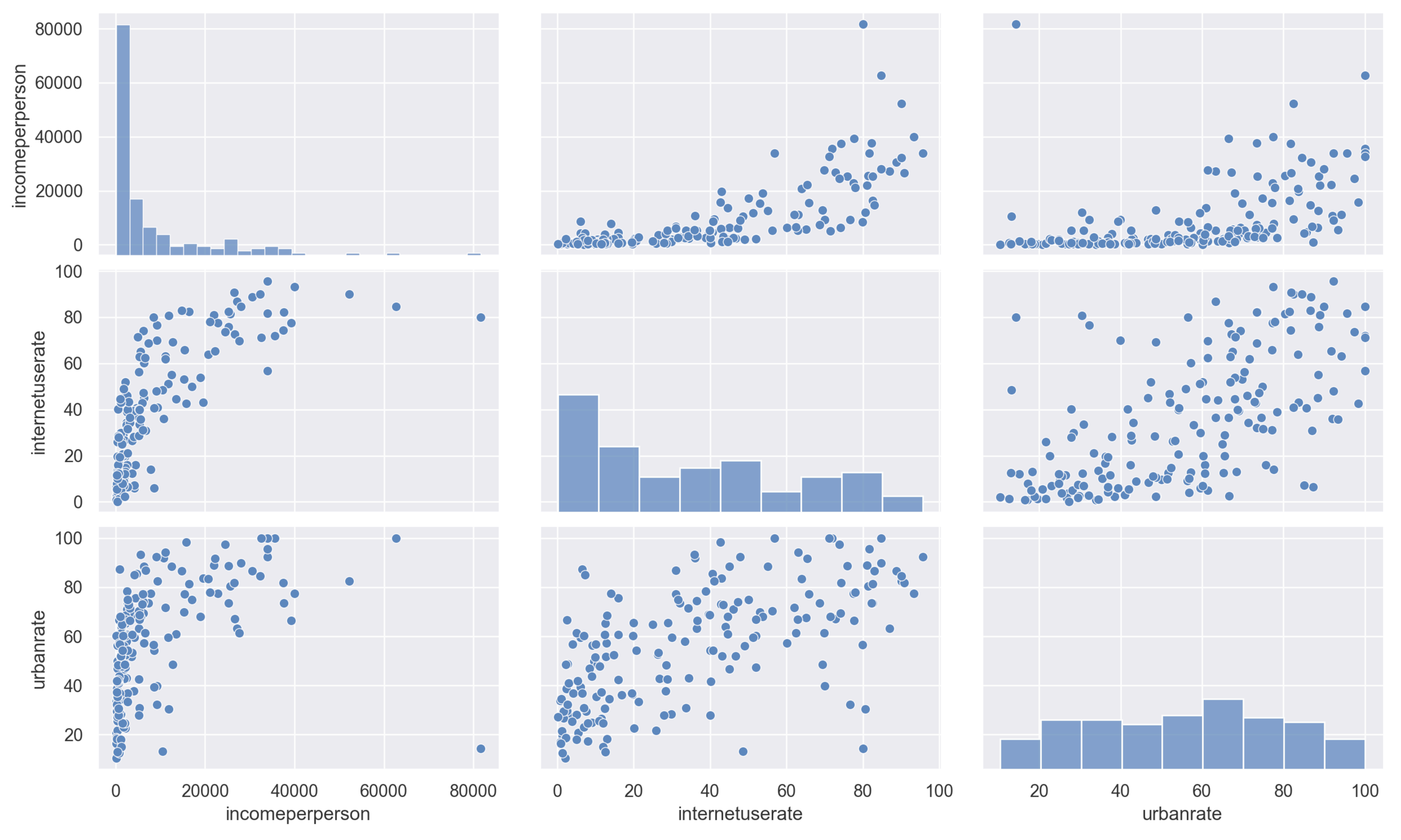
Week 9

Regression Example

- Get internet usage data from kaggle

```
inter = pd.read_csv('internet.csv')
inter.dropna(inplace=True)
seaborn.pairplot(inter)
plt.show()
```

Regression Example



Regression Example

- Most prominent is the relation income —> internet use
    - But this is definitely not linear
        - Try out square-root and cubic-root

```
inter['sqr'] = inter.incomeperperson**0.5
inter['cr'] = inter.incomeperperson**(1/3)
```

Regression Example

- We try out different models

```
model = ols('internetuserate ~ incomeperperson + sqr + cr ',
inter).fit()
print(model.summary())
a, b1, b2, b3 = (model.params)
```

Regression Example

- With all data

```
model = ols('internetuserate ~ incomeperperson + sqr + cr +
urbanrate', inter).fit()
print(model.summary())
print(model.params)
a, b1, b2, b3, c = (model.params)
print('b1', b1, b1*inter.incomeperperson.std()/
inter.internetuserate.std())
print('b2', b2, b2*inter.sqr.std()/inter.internetuserate.std())
print('b3', b3, b3*inter.cr.std()/inter.internetuserate.std())
print('c', c, c*inter.urbanrate.std()/inter.internetuserate.std())
```

Regression Example

- We look at the change of R^2 and the relative slopes to decide that

```
b1 -0.0014352058731803116 -0.6401231587766388
b2 0.7060138860595546 1.4177729717900212
b3 0.07024055095535309 0.021435848641083485
c 0.04632491019563843 0.03902714038562757
```

- only first and second variable (income and square-root of income) are important

Regression Example

- We fit the model and obtain the parameters

```
a -5.853396168966507
b1 -0.0015578988847916358 -0.6948460497676745
b2 0.7546950130884635 1.5155313693239216
```

- We create an interval and add a prediction column to the data frame

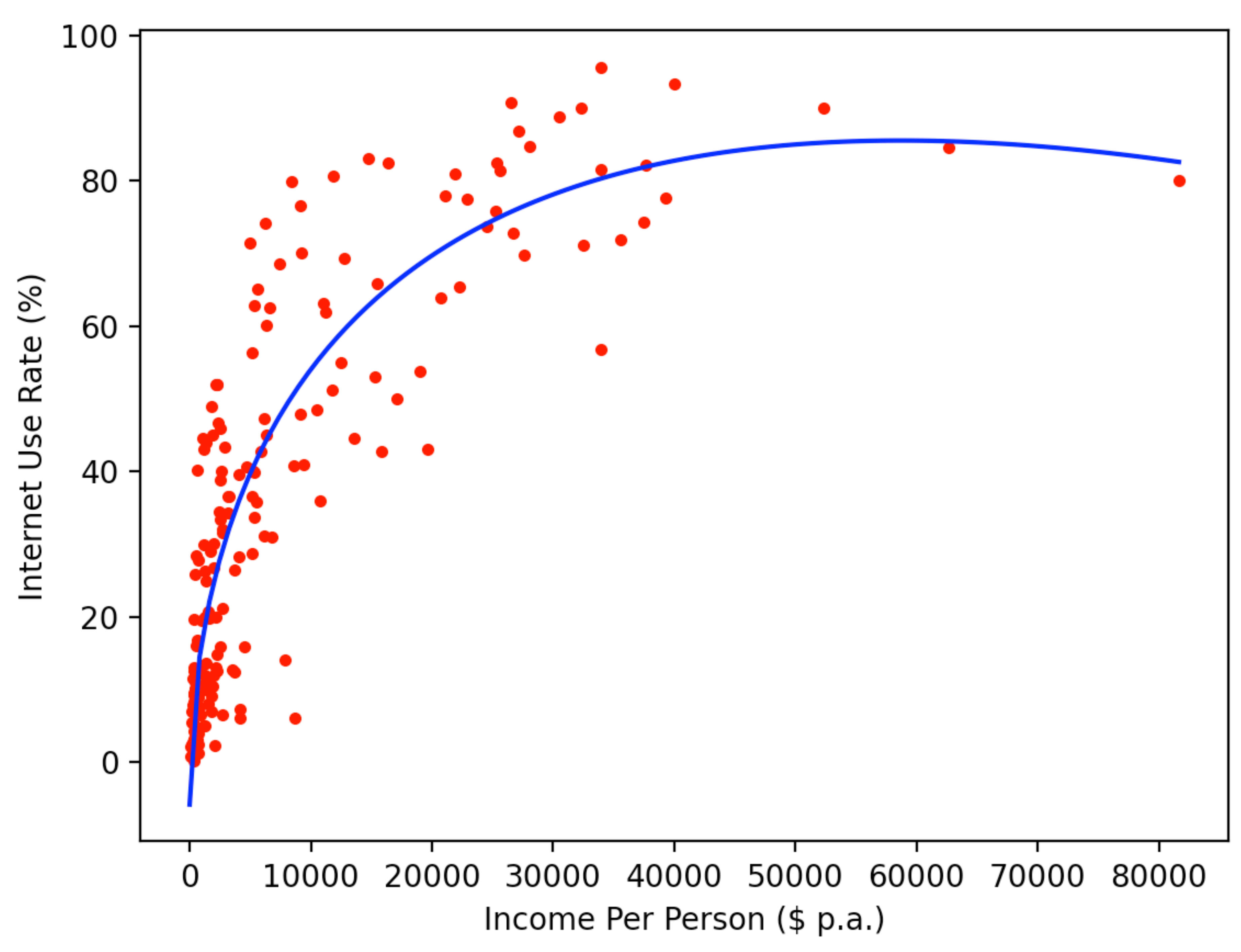
```
x = np.linspace(0, np.max(inter.incomeperperson),101)
pred =a+b1*x+b2*x**(1/2)
```

Regression Example

- And then display the result

```
plt.plot(inter.incomeperperson, inter.internetuserate, '.r')
plt.plot(x, pred , 'b-')
plt.xlabel('Income Per Person ($ p.a.)')
plt.ylabel('Internet Use Rate (%)')
plt.title=('Internet Usage')
plt.show()
```

Regression Examples



Regression Example

- Criticizing the model:
  - We predict for high income countries, that internet usage goes down
    - This is somewhat counterintuitive and based on a couple of outliers.

Regular Expression and Web Scraping

Important Preliminaries

- On your own machine:
    - Install pip3 (the python 3 version)
        - You can invoke pip3 also by python3 -m pip
    - Then install a number of packages:
        - beautifulsoup4
            - sudo python3 -m pip install beautifulsoup4
        - requests

Scraping and Crawling

- Both involve automatic ('bot') access to a web-site
- Crawling tries to find and process all the information on all pages of the website
  - Typically used by search engines
- Scraping
  - Used to obtain data contained in certain web-pages

Legal and Ethical Issues

- Web-scrapping is sometimes considered a threat
  - Because it creates real problems
  - Because it accesses data for use against the business interests of the web service provider

Legal and Ethical Issues

- Web-scraping can run afoul of:
  - Existing and future laws
    - In the US:
      - Computer Fraud and Abuse Act, Digital Millennium Copyright Act,
  - Terms of Use / Breach of Contract e.g. those in robots.txt
  - Copyright
  - ...

Legal and Ethical Issues

- robots.txt gives conditions for automatic crawling
  - No crawling:
    ```
    User-agent: *
    Disallow: /
    ```
  - All crawling allowed:
    ```
    User-agent: *
    Disallow:
    ```
  - Block twitbot from crawling the indicated directory
    ```
    User-agent: twitbot
    Disallow: /mysecrets/
    ```

Legal and Ethical Issues

- robots.txt
  - Needs to be called that (not Robots.txt)
  - Needs to be placed in the top-level of the hierarchy
  - needs to be publicly available
  - subdomains will have to use separate robots files
  - Can be used to provide a sitemap for crawlers (so that search engines will show your content)
    - Sitemap: `https://www.mysite.com/sitemap.xml`

Legal and Ethical Issues

- Aggressive scraping (and crawling) can become a Denial of Service Attack
  - Server busy to answer scraping demands and cannot serve other traffic
  - robots.txt can specify a desired back-off interval
    - In general: do not access web-pages on a site without an interval of at least 10 seconds

Legal and Ethical Issues

- Many sites provide APIs in order to allow users to make bulk-downloads of data
  - This usually means they do not want to have their site scraped, so they offer a simpler alternative

Legal and Ethical Issues

- Raw data is not protected by copy-right
- Exceptions can arise when scraping is used to obtain the same functionality as the original site
- Scraping needs to be done at a low level of intensity
- Using an agent that sends identifying information with each request is useful
    - Security pouring over logs can be put at ease with an explanation

Legal and Ethical Issues

- Websites are free to ban robots by using a black-list for IP addresses
  - Commercial crawling solutions exists that circumvent banning
    - Imitate human user behavior
    - Use many different IP addresses
    - Automatic throttling of requests
- The need and the existence of these automated crawlers show that:
  - Scraping is in a legal and ethical gray-zone

Techniques

- To download data from a website and prepare it for processing
    - We need to access the website
    - We need to find the data on the website and put it into a structure we can use
- Before we code, we need to first understand the source of the website
- After we obtained the data, we need to store it in a reasonable format

Understanding Web Sites

- Access the target website
- Use the developer tools or view the source
  - Browser dependent

Understanding Web Sites

- Milwaukee police maintains a website with current call data
  - https://itmdapps.milwaukee.gov/MPDCallData/
  - Goal is to download this data
  - Use the "Show Source Functionality" of your browser on the website

Understanding Web Sites

```
<!DOCTYPE HTML>

<html>

<head>
    <title>Milwaukee Police Department: Call for Service</title>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <meta name='viewport' content='width=device-width, initial-scale=1'>
    <link rel='stylesheet' href='/ItmdScripts/css/redesign.css' type='text/css'/>
    <link rel='stylesheet' href='/ItmdScripts/css/city-various.css' type='text/css'/
>
    <script src='/ItmdScripts/js/jquery.min.js'></script>
    <script src='/ItmdScripts/js/message.js'></script>
    <script src='/ItmdScripts/js/mil-default.js'></script>
</head>
<body>
    <div id='bg-div'>
        <div data-role='page' class='main'>
            <div data-role='header' class='redesign-header'>
                <a id='lnk-citylogo' href='http://city.milwaukee.gov'>
                    <img alt='City of Milwaukee' src='//itmdapps.milwaukee.gov/
templates/2014/city2013_logo.png'/>
                </a>
                <div class='city-title'>Official Website of the City of Milwaukee</
div>
                <div id='city-navbar-div'>
                    <div style='position: relative'>
```

Understanding Web Sites

- Identify the data that we would like to extract
    - In this case, data in a table

```
<tr style='border-style: none; border-collapse: collapse;'>
                            <td style='border: 1px solid black;
border-collapse: collapse;'>201731019</td>
                            <td style='border: 1px solid black;
border-collapse: collapse;'>06/21/2020 11:54:25 AM</td>
                            <td style='border: 1px solid black;
border-collapse: collapse;'>6000 W SILVER SPRING DR,MKE</td>
                            <td style='border: 1px solid black;
border-collapse: collapse; text-align: center;'>4</td>
                            <td style='border: 1px solid black;
border-collapse: collapse;'>PATROL</td>
                            <td style='border: 1px solid black;
border-collapse: collapse;'>Assignment Completed</td>
                        </tr>
```

Understanding Websites

- Before we start downloading websites, let's first understand them
  - Each web browser has a way to view the source of a website
  - On Chrome, use Developer -> View Source
    - Easiest tool for web development

Accessing Web Sites

- Selenium: Module for automatic web application tests
  - Automatically click links, pretend to be a certain browser, etc
  - Useful when data is accessed after ajax requests
  - Needs some downloads

Accessing a Web Site

- Scrapy:
  - Framework to run scraping and web crawling
  - Developed by web-aggregation and e-commerce company Mydeco
  - Maintained by Scrapinghub
  - Interlaced with a commercial offering

Accessing websites

- Requests
  - Simple and basic translator for making url requests
    - `r = requests.get(address)`
  - Variable `r.content` now contains the contents of the web page (as a binary string)
  - Variable `r.text` contains the contents as a string
    - Requests will guess the encoding
    - But you can set the encoding with
      - `response.encoding = 'utf-8'`

Accessing websites

- Requests
  - Can use `r.headers` to obtain a dictionary-like object with various header values
  - Can use query string in requests:
    - Example:

```
requests.get('https://api.github.com/search/rep',
            params=[('q', 'requests+language:python')])


requests.put('https://httpbin.org/put',
data={'key':'value'})
```

Accessing Websites

- Use regular expression (just a little bit)
- Use beautiful soup (html parser)
- Use requests

Regular Expressions

Python

Why

- A frequent programming task is “filtering”
  - Retain only those records that fit a certain pattern
  - Typical part of big data and analytics applications
- Example for text processing

Why

- Whenever you deal with text processing
  - Think about whether you want to use regular expressions

Why

- Regular Expressions are a theoretical concept that is well understood
- Many programming languages have a module for regular expressions
    - Usually, very similar syntax and semantics
- We can use ad hoc solutions, but regular expressions are almost always faster

How

- Usually, we want to compile a regular expression
  - This allows for faster scanning
    - Compilation cost time
      - But usually amortized very quickly
- Python regular expressions are in module re
  - Use `p=re.compile('?')`
    - Where the question mark is the search string

How

- A Python regular expression is a string that defines the search
- The string is compiled
- After compilation, a match, search, or findall is performed on all strings
    - The output is None if the regular expression is not matched
    - Otherwise, depending on the function, it provides the parts of the string that match

A first example

- In a regular expression, most characters match themselves
    - Unless they are "meta-characters" such as *, \, ^
- E.G.: Find all lines in "alice.txt" with a double hyphen
- Regular expression is `'--'`
- Read in all lines of the text file, find the ones that match
    - Need to use search, because match only matches at the beginning of a string

A first example

```
import re

p = re.compile('--')

def match1():
    with open("alice.txt") as infile:
        line_count = 0
        for line in infile:
            line_count+=1
            line = line.strip()
            if p.search(line):
                print(line_count, line)
```

- Import re
- Compile the regular expression
- Match lines with .search()

Using raw strings

- A raw string is a string preceeded with a letter r:
  - `print(r'Hello World')`
- The difference to a normal string is that the escape character always means the escape character itself.
  - `print(r'\tHello')` prints out `\tHello`
  - `print('\tHello')` prints out `Hello` after a tab.
- This can be very useful because we might on occasion have to escape the escape character several times.

Matching

- Characters are the easiest to match
  - Find all words in lawler.txt (a large list of English words) with a double "oo"
    - Just change the expression
    -

```
import re

p = re.compile('oo')

def match1():
    with open("lawler.txt") as infile:
        line_count = 0
        for line in infile:
            line_count+=1
            line = line.strip()
            if p.search(line):
                print(line_count, line)
```

Matching

- Letters and numbers match themselves
- But are case sensitive
- Punctuation marks often mean something else.

Matching

- Square brackets [ ] mean that any of the enclosed characters will do
  - Example: [ab] means either 'a' or 'b'
- Square brackets can contain a range
  - Example: [0-5] means either 0, 1, 2, 3, 4, or 5
- A caret ^ means negation
  - Example: [^a-d] means neither 'a', 'b', 'c', nor 'd'

Self Test

- Find all lines in a file that have a double 'e'

Self Test Solution

```
import re

p = re.compile(r'ee')

def match_ee(filename):
    with open(filename) as infile:
        for line in infile:
            if p.search(line):
                print(line.strip())
```

Self Test 2

- Find all lines in a file that have a double-'ee' followed by a letter between 'l' (el) and 'n'

Self Test 2 Solution

The only difference is in the regular expressions where we have now a range of letters.

```
import re

p = re.compile(r'ee[l-m]')

def match_ee(filename):
    with open(filename) as infile:
        for line in infile:
            if p.search(line):
                print(line.strip())
```

Matching: Wild Cards

- Wild Card Characters
  - The simplest wild card character is the period / dot: “.”
  - It matches any single character, but not a new line
  - Example: Find all English words using Lawler.txt that have a patterns of an “a” followed by another letter followed by “a”
  - Solution: Use `p = re.compile('a.a')`

Matching: Wild Cards

- If you want to use the literal dot `'.'` you need to escape it with a backslash
- Example: To match “temp.txt” you can use `'t...\.txt'`
  - This matches any file name that starts with a t, has three characters afterwards, then a period, and then txt.

Matching: Repetitions

- The asterisks repeats the previous character zero or more times
  - Example: `'\.[a-z]*'` looks for a period, followed by any number of small letters, but will also match the simple string `'.'`
- The plus sign repeats the previous character one or more times.
  - Example: `'uni[a-z]+y'` matches a string that starts with 'uni' followed by at least one small letter and terminating with 'y'
    - This is difficult to read, as the + looks like an operation

Matching: Repetitions

- Braces (curly brackets) can be used to specify the exact number of repetitions
  - 'a{1:4}' means one, two, three, or four letters 'a'
  - 'a{4:4}' means exactly four letters 'a'

Self Test

- Print all file names in a directory that look like a Python file.
    - Notice that ".py" is not a valid Python file. There must be something before the dot.

Self Test Solution

```
def get_python(dir_name):
    python = re.compile('.+\\.py')
    lista = os.listdir(dir_name)
    for name in lista:
        if python.match(name):
            print(name)
```

Matching

- \w stands for any letter (small or capital) or any digit
- \W stands for anything that is **not** a letter or a digit
- Example: Matching "n"+non-letter/digit+"t"

```
"Speak English!" said the Eaglet. "I don't know the meaning of half
They were indeed a queer-looking party that assembled on the bank
```

- `p = re.compile('n\\Wt')`
  - We need to double escape the backslash using normal Python strings
- `p = re.compile(r'n\Wt')`
  - Or use a "raw string" (with an "r" before the string)
  - In a raw string, the backslash is always a backslash

Matching

- \s means a white space, newline, tab
- \S means anything but a white space, newline, or tab
- \d matches a digit
- \t matches a tab
- \r matches a return

Regular Expression Functions

- Once compiled a regular expression can be used with
  - `match()` matches at the beginning of the string and returns a match object or None
  - `search()` matches anywhere in the string and returns a match object or None
  - `findall()` matches anywhere in the string and **does not return a match object**

Match Objects

- A match object has its own set of methods
  - `group()` returns the string matched by the regular expression
  - `start()` returns the starting position of the matched string
  - `end()` returns the ending position
  - `span()` returns a tuple containing the (start, end) positions of a match

Regular Expression Gotcha

- Regular expression matching is **greedy**
  - Prefers to match as much of the string as it possibly can
- Example:

```
p3 = re.compile(r'.+\.py')
print( p3.search("This file, hello.py and this file
world.py are python files"))
```

- Prints out

```
<re.Match object; span=(0, 42), match='This file,
hello.py and this file world.py'>
```

Non-Greedy Matching

- We can use the question mark qualifier to obtain a non-greedy match.
  - `p = re.compile('o.+?o')`
- Finds all non-overlapping, minimal instances

Advanced Topics

- In this module we only scratched the surface.
- There is excellent online documentation if you need more information
- But this should be sufficient to do simple tasks such as data cleaning and web scraping

Webscraping with BeautifulSoup

Thomas Schwarz, SJ

Beautiful Soup

- Module developed for parsing web-pages
  - Current version is called **bs4**
  - `from bs4 import BeautifulSoup`

Beautiful Soup Installation

- Easy installation with pip
  - Just remember that you need to install it for the correct Python version

HTML in Five Minutes

- HTML is a **markup language**
  - Tags < > are used to delimit elements
- HTML documents start out and end with an <html> </html> tag
- HTML documents consists of two parts:
  - Head: <head> </head>
  - Body: <body> </body>
    - Head: Information on the page
    - Body: The page itself

HTML in Five Minutes

- Basic html elements:
  - Text header <h1></h1>, ... <h6></h6>
  - Paragraphs <p></p>
  - Links <a> anchors
  - Images <im> </im>
  - Lists <ul>
  - Dividers <div>
  - Spans <span>

HTML in Five Minutes

- Often, tags have metadata embedded.
  - Example:
  - `<a href="https://tschwarz.mscs.mu.edu">Schwarz </a>`
    - A link with a property href set
  - An ordered list using capital letters as numbers
    - `<ol type = "A">`

Beautiful Soup Parser

- Start out by creating a Beautiful Soup object
  - Need to have a parser attached
    - Standard is the html parser

```
import requests
from bs4 import BeautifulSoup

r = requests.get(url)
soup = BeautifulSoup(r.content, 'html.parser')
```

Beautiful Soup Parser

- We can use prettify( ) in order to find print out the contents of the beautiful soup obejct.
  - Step 1: Import the modules

```
from bs4 import BeautifulSoup
from requests import get
```

  - Step 2: Scrape

```
def scrape():
    return get('https://tschwarz.mscs.mu.edu')
```

  - Step 3: Display the contents

```
def display():
    soup = BeautifulSoup(scrape().content,
            features = 'html.parser')
    print(soup.prettify())
```

Beautiful Soup Parser

- The 'html.parser' comes with Python
- There are a number of other parsers that can be installed
- See the BeautifulSoup/bs4 documentation

BeautifulSoup Objects

- An html tag defines an html element
  - We can access tag elements from within BeautifulSoup
  - The first tag element can be accessed just by using the tag
    - Example: Getting the first li tag on my website:

```
import requests
from bs4 import BeautifulSoup

soup = BeautifulSoup(ts.content, 'html.parser')
print(soup.li)
```

Beautiful Soup Objects

- HTML tags have names
  - <a> (anchor) tag has name a
  - <p> (paragraph) tag has name p
- HTML tags have attributes
  - class, id, style, ...

Beautiful Soup Objects

- Getting the name of a tag:

```
import requests
from bs4 import BeautifulSoup

soup = BeautifulSoup(ts.content, 'html.parser')
li_tag = soup.li
print(li_tag.name)    # prints out   li
```

- We could actually change the name of a tag and thereby beautiful soup parse tree

Beautiful Soup Objects

- Getting attributes of a tag
  - In the example, the li - tag has an anchor inside.
    - We can get to the anchor
      - The attributes are in a dictionary

Beautiful Soup Objects

- Example `print(li_tag.a)`
- Prints out

```
<a class="tab_active" href="index.html"
target="_self">Home</a>
```

- Attributes are in a dictionary:

```
>>> print(li_tag.a.attrs)
{'class': ['tab_active'], 'href': 'index.html',
 'target': '_self'}
```

- and accessible directly

```
>>> print(li_tag.a['class'])
['tab_active']
```

Beautiful Soup Objects

- To get to the text in a tag, use .string

```
>>> print(li_tag.a.string)
Home
```

Searches in Beautiful Soup

- To search within a BeatifulSoup object, we can use
  - find
    - Only finds first occurrence
  - find_all
    - Returns a list of occurrences

Searches in Beautiful Soup

- Find can use
  - a tag , e.g. an anchor

    ```
    soup.find('a')   soup.find(name = 'a')
    ```

  - a text string or a regular expression
    - Careful: You are looking for the exact string.

```
>>> mke = soup.find(text = re.compile('Milwaukee'))
>>> mke
'Milwaukee Police Department: Call for Service'
>>> mke = soup.find(text = 'Milwaukee')
>>> print(mke)
None
```

Searches in Beautiful Soup

- Find can use attributes of tags
  - Generic: Use attrs parameter with a dictionary

```
>>> footer = soup.find(attrs={'class' : "footer"})
>>> footer
<div class="footer" data-role="footer"><ul><li><a
href="http://city.milwaukee.gov/Mayor">Mayor Tom Barrett</
a></li><li><a href="http://city.milwaukee.gov/
CommonCouncil">Common Council</a></li></ul><ul><li><a ...
```

Searches in Beautiful Soup

- Can use find with a function
    - Function is boolean, i.e. returns True or False

Searches in Beautiful Soup

- find_all works like find, but returns a list of results
- In addition, `limit=n` limits the list to the first results

Case Study: MPD

- Go to https://itmdapps.milwaukee.gov/MPDCallData/ and save the file
  - We do not want to upset the police

Case Study: MPD

- First, we use beautiful soup to show us the file:

```
def prob3():
    with open('mpd.html') as mpd:
        soup = BeautifulSoup(mpd, 'html.parser')
        print(soup.prettify())
```

- This is just a nicer version of the html file
  - The call data is in a single table

Case Study: MPD

- Now let's find all tables: Look for tr

```
def prob4():
    with open('mpd.html') as mpd:
        soup = BeautifulSoup(mpd, 'html.parser')
        results = soup.find_all('tr')
        for item in results:
            print('an item:')
            print(item)
            print()
```

Case Study: MPD

- This gives us lots of tables, some belonging to navigation and some belonging to what we are looking for

```
an item:
<tr style="border-style: none; border-collapse: collapse;">
<td style="border: 1px solid black; border-collapse: collapse;">201731676</td>
<td style="border: 1px solid black; border-collapse: collapse;">06/21/2020 04:56:37 PM</td>
<td style="border: 1px solid black; border-collapse: collapse;">2423 S 6TH ST,MKE</td>
<td style="border: 1px solid black; border-collapse: collapse; text-align: center;">2</td>
<td style="border: 1px solid black; border-collapse: collapse;">TRBL W/SUBJ</td>
<td style="border: 1px solid black; border-collapse: collapse;">Advised</td>
</tr>
```

- The good stuff is the third item in the list

Case Study: MPD

- First, let's restrict ourselves to the good stuff

```
def prob5():
    with open('mpd.html') as mpd:
        soup = BeautifulSoup(mpd, 'html.parser')
        results = soup.find_all('tr')[2:]  #use slicing
        return results
```

Case Study: MPD

- Then inside these results, let's look for the columns (td)

```
def prob6():
    with open('mpd.html') as mpd:
        soup = BeautifulSoup(mpd, 'html.parser')
        results = soup.find_all('tr')[1:]
        for r in results:
            print('\n')
            for e in r.find_all('td'):
                print(e)
```

Case Study: MPD

- Now we can take out the contents
  - Strategy:
    - For each row create a dictionary
    - Use

```
from dateutil.parser import parse
```

      - to parse the date time

Case Study: MPD

```
def prob7():
    findall = [ ]
    with open('mpd.html') as mpd:
        soup = BeautifulSoup(mpd, 'html.parser')
        results = soup.find_all('tr')[2:]
        for r in results:
            entries = [e.contents[0] for e in
r.find_all('td')]
            datetime = parse(entries[1])
            dicti = { 'id': entries[0],
                      'datetime': datetime,
                      'address': entries[2],
                      'district': entries[3],
                      'descr': entries[4],
                      'status': entries[5]}
            findall.append(dicti)
    return findall
```

Case Study: MPD

- Finally, can create a data frame

```
data = pd.DataFrame(prob7())
```

```
>>> data.head()
          id            datetime  ...            descr               status
0  201731692 2020-06-21 17:04:39  ...  TRAFFIC HAZARD  Assignment Completed
1  201731630 2020-06-21 17:03:42  ...         ACC PDO   Service in Progress
2  201731573 2020-06-21 17:03:09  ...  FAMILY TROUBLE               Advised
3  201731601 2020-06-21 17:02:26  ...          THREAT   Service in Progress
4  201731683 2020-06-21 17:02:05  ...          ACC PI   Service in Progress
```

Case Study: Finding Links

- Let's use request in order to do some web-navigation
  - Target is my web-site:
    - https://tschwarz.mscs.mu.edu/Classes/
  - We just beat up on this one

Case Study: Finding Links

- First step:
  - Store addresses in global constant
    - Use the class Gl trick

```
class Gl:
    site = 'https://tschwarz.mscs.mu.edu/Classes'
    file = 'classes.html'
    regex_headers = re.compile(r'<h.>.*?</h.>')
    regex_links = re.compile(r'href=".*?"')
    regex_div = re.compile(r'<div.*?>')
```

Case Study: Finding Links

- Second step:
  - Download the page

```
def get_links(site=Gl.site, file=Gl.file):
    webpage = requests.get('/'.join([site, file])).text
```

Case Study: Finding Links

- Third step:
  - Find all links
    - Links use the <a href = … > construct

```
class Gl:
    regex_links = re.compile(r'href=".*?"')
```

Case Study: Finding Links

- Third step:
    - This will give us exactly the links as a list

```
['href="../style.css"', 'href="../style_extra.css"',
'href="../index.html"', 'href="../cv.html"',
'href="publications.html"', 'href="classes.html"',
'href="PDS/index.html"', 'href="Algo2020/index.html"',
'href="AlgoF2020/index.html"', 'href="PDS/index.html"'
'href="Ahmedabad2019/Python.html"',
'href="Ahmedabad2019/index.html"', 'href="Mumbai2019/
index.html"', 'href="Mumbai2020/index.html"',
'href="AhmedabadDataAtScale/index.html"',
'href="COSC1010F2019/index.html"', 'href="COSC1010/
index.html"', 'href="Algorithms/index.html"',
'href="DataAtScale/index.html"']
```

Case Study: Finding Links

- We cut out the beginning '`html="`' and the ending '`"`'

```
def get_links(site=Gl.site, file=Gl.file):
    webpage = requests.get('/'.join([site, file])).text
    lista = Gl.regex_links.findall(webpage)
    for element in lista:
        element = element[6:-1]
        print(site+'/'+element)
```

Case Study: Finding Links

- We could now add all of the resulting websites into a list
  - Which we then could crawl, if we wanted to

```
def get_links(site=Gl.site, file=Gl.file):
    webpage = requests.get('/'.join([site, file])).text
    lista = Gl.regex_links.findall(webpage)
    result = [ ]
    for element in lista:
        element = element[6:-1]
        result.append(site+'/'+element)
    return result
```

Case Study: Downloading images from a web-site

- Google has an API googlesearch that allows you to find addresses of pages with a query
- Used this to obtain a random target
  - Please be nice and find a different target because they might be paying per access to page

Case Study:
Downloading images from a web-site

- Import the tools of the trade and set the target url

```
from bs4 import BeautifulSoup
import requests
import urllib.request

url = "http://thehibbitts.net/troy/photo/birds/
sandhill_crane.htm"
```

Case Study: Downloading images from a web-site

- Use beautiful soup in order to find all instances of an img tag:

```
url = "http://thehibbitts.net/troy/photo/birds/
sandhill_crane.htm"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")
aas = soup.find_all("img")
```

Case Study: Downloading images from a web-site

```
>>> aas
[<img height="600" src="sandhill.cranes.tx.archer.17.1.3956.jpg"
width="900"/>, <img height="600"
src="sandhill.cranes.tx.archer.17.1.3986.jpg" width="900"/>, <img
height="600" src="sandhill.cranes.tx.archer.17.1.3997.jpg"
width="900"/>, <img height="558"
src="sandhill.crane.mi.delta.16.1.0145.jpg" width="834"/>, <img
height="558" src="sandhill.crane.mi.delta.16.1.0139.jpg"
width="834"/>, <img height="600"
src="sandhill.cranes.wi.jackson.13.1.0095.jpg" width="900"/>,
<img height="600" src="sandhill.cranes.wi.jackson.13.1.0086.jpg"
width="900"/>, <img border="0" height="555"
src="sandhill.cranes.tx.uvalde.10.1a.jpg" width="835"/>, <img
border="0" height="554" src="sandhill.cranes.tx.uvalde.10.1b.jpg"
width="834"/>, <img border="0" height="554"
src="sandhill.cranes.tx.uvalde.10.1c.jpg" width="834"/>, <img
border="0" height="602"
src="sandhill.cranes.tx.uvalde.07.1a1.jpg" width="837"/>, <img
border="0" height="577" src="sandhill.cranes.tx.uvalde.07.1a.jpg"
width="838"/>]
```

Case Study:
Downloading images from a web-site

- Convert them to strings and divide them into components
- Then extract the url

```
for line in aas:
    line = str(line)
    components = line.split()
    for component in components:
        if component.startswith('src'):
            image_url = component[5:-1]
```

Case Study: Downloading images from a web-site

- Then get the image and save it as a file under the same name

```
imurl = "http://thehibbitts.net/troy/photo/
birds/"+image_url
            image = requests.get(imurl).content
            with open(image_url, 'wb') as file:
                file.write(image)
```

Case Study:
Downloading images from a web-site

- Result:
  - You downloaded all the images

Summary

- If data is published on the web:
  - First, see whether the data is available through an API
    - Administrators get annoyed if people scrape unnecessarily
  - If data is available only as html data:
    - Be careful in making large number of requests.
      - This can get you banned / blacklisted
    - You might get a complaint from the legal department
      - Which is usually not valid unless you exploit for commercial nature