Obtendo dados

Para ser um cientista de dados, você precisa de dados. Na verdade, como cientista de dados, você gastará uma grande parte do tempo adquirindo, limpando e transformando dados.

stdin e stdout

Se você executar seus scripts Python na linha de comando, poderá canalizar dados através deles usando sys.stdin e sys.stdout.

Por exemplo, aqui está um script que lê linhas de texto e mostra as que correspondem a uma expressão regular:

In [1]:
```
# egrep.py
import sys, re
# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]
# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

E aqui está um outro script que conta as linhas que recebe e depois escreve a contagem:

In [2]:
```
# line_count.py
import sys
count = 0

for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print(count)
```

```
0
```

Você poderia então usá-los para contar quantas linhas de um arquivo contêm números. No Windows, você usaria:

In [ ]:
```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

enquanto que em um sistema Unix você usaria:

In [7]:
```
!cat uscensus.txt | python egrep.py "[0-9]" | python line_count.py
```

```
4665
```

Da mesma forma, aqui está um script que conta as palavras em sua entrada e escreve as mais comuns:

In [ ]:
```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1) # non-zero exit code indicates error

    counter = Counter(word.lower() # lowercase words
                      for line in sys.stdin #
                      for word in line.strip().split() # split on spa
ces
                      if word) # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

after which you could do something like:

In [ ]:
```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193 the
51380 and
34753 of
13643 to
12799 that
12560 in
10263 he
9840 shall
8987 unto
8836 for
```

In [10]:
```
!cat uscensus.txt | python most_common_words.py 10
```

```
919     to
768     of
663     percent
561     total
409     and
408     or
408     years.........................................
307     occupied
307     persons
307     housing
```

Lendo arquivos

Você também pode ler explicitamente e gravar em arquivos diretamente no seu código. O Python torna o trabalho com arquivos bastante simples.

Noções básicas de arquivos de texto

O primeiro passo para trabalhar com um arquivo de texto é obter um objeto de arquivo usando open:

In [ ]:
```
# 'r' means read-only
file_for_reading = open('reading_file.txt', 'r')

# 'w' is write - will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append - for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Como é fácil esquecer de fechar seus arquivos, você deve sempre usá-los em um bloco "with" pois ao final eles serão fechados automaticamente:

In [ ]:
```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

Se você precisar ler um arquivo de texto inteiro, basta fazer uma iteração nas linhas do arquivo usando:

In [ ]:
```
starts_with_hash = 0
with open('input.txt','r') as f:
    for line in file: # look at each line in the file
        if re.match("^#",line): # use a regex to see if it starts w
ith '#'
            starts_with_hash += 1 # if it does, add 1 to the count
```

Cada linha que você recebe dessa maneira termina em um caractere de nova linha, então você frequentemente vai querer limpar antes de fazer qualquer coisa com ela.

Por exemplo, imagine que você tenha um arquivo cheio de endereços de e-mail, um por linha, e que você precise gerar um histograma dos domínios.
Uma boa maneira é apenas pegar as partes dos endereços de e-mail que vêm depois do @.

In [11]:
```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

In [12]:
```
domain_counts
```

Out[12]:
```
Counter({'gmail.com': 5, 'hotmail.com': 2})
```

Arquivos Delimitados

O arquivo hipotético de endereços de e-mail que acabamos de processar tinha um endereço por linha. Com mais frequência, você trabalha com arquivos com muitos dados em cada linha. Esses arquivos geralmente são separados por vírgulas ou separados por tabulações. Cada linha possui vários campos, com uma vírgula (ou uma tabulação) indicando onde um campo termina e o próximo campo é iniciado.

Isso começa a ficar complicado quando você tem campos com vírgulas e tabulações e novas linhas neles (o que você inevitavelmente faz). Por esse motivo, é quase sempre um erro tentar analisá-las você mesmo.

Em vez disso, você deve usar o módulo csv do Python (ou a biblioteca pandas).
É sempre uma boa prática trabalhar com arquivos csv no modo binário, incluindo um b após r ou w.

Se o seu arquivo não tiver cabeçalhos, use csv.reader para iterar nas linhas, cada uma delas será uma lista dividida apropriadamente. Por exemplo, se tivéssemos um arquivo de preços de ações delimitado por tabulações:

In [ ]:
```
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34
```

nós poderíamos processá-los com:

In [18]:
```
import csv
with open('tab_delimited_stock_prices.txt', 'r') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        print(date, symbol, closing_price)
```

```
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34
```

Se o seu arquivo tiver cabeçalhos:

In [15]:
```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

```
  File "<ipython-input-15-cdda1635a397>", line 1
    date:symbol:closing_price
               ^
SyntaxError: invalid syntax
```

você pode ignorar a linha de cabeçalho (com uma chamada inicial para reader.next ()) ou **obter cada linha como um dict (com os cabeçalhos como chaves) usando csv.DictReader**:

In [21]:
```
with open('colon_delimited_stock_prices.txt', 'r') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        print(date, symbol, closing_price)
```

```
6/20/2014 AAPL 90.91
6/20/2014 MSFT 41.68
6/20/2014 FB 64.5
6/19/2014 AAPL 91.86
6/19/2014 MSFT 41.51
6/19/2014 FB 64.34
```

Mesmo que seu arquivo não tenha cabeçalhos, você ainda poderá usar o DictReader passando as chaves como um parâmetro de nome de campo.
Você também pode gravar dados delimitados usando csv.writer:

In [20]:
```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }
with open('comma_delimited_stock_prices.txt','w') as f:
    writer = csv.writer(f, delimiter='\t')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

csv.writer fará a coisa certa se seus campos tiverem vírgulas neles. Seu próprio escritor programado à mão provavelmente não vai fazer corretamente. Por exemplo, se você tentar:

In [22]:
```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'w') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many comm
as in it!
        f.write("\n") # row might have newlines as well!
```

Você vai acabar com um arquivo csv que se parece com:

In [ ]:
```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

In [23]:
```
!cat bad_csv.txt
```

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

e que ninguém nunca será capaz de entender.

Raspando (scraping) a web

Outra maneira de obter dados é raspando-os de páginas da web. Buscar páginas da web é bem fácil; obtendo informação estruturada significativa fora deles menos.

HTML e seu entendimento

Páginas na Web são escritas em HTML, em que o texto é (idealmente) marcado em elementos e seus atributos:

In [ ]:
```
<html>
<head>
<title>A web page</title>
</head>
<body>
<p id="author">Joel Grus</p>
<p id="subject">Data Science</p>
</body>
</html>
```

Em um mundo perfeito, onde todas as páginas da Web são marcadas semanticamente em nosso benefício, poderíamos extrair dados usando regras como "encontre o elemento < p> cujo id está sujeito e retorne o texto que ele contém". No mundo real HTML não é geralmente bem formado, muito menos anotado. Isso significa que precisaremos de ajuda para entender isso.

Para obter dados de HTML, usaremos a biblioteca BeautifulSoup, que constrói uma árvore a partir dos vários elementos em uma página da Web e fornece uma interface simples para acessá-los. Enquanto escrevo isso, a última versão é Beautiful Soup 4.3.2 (pip install beautifulsoup4), que é o que vamos usar. Também usaremos a biblioteca de solicitações (solicitações de instalação de pip), que é uma maneira muito mais agradável de fazer solicitações HTTP do que qualquer outra que esteja incorporada ao Python.

O analisador de HTML interno do Python não é tão tolerante, o que significa que nem sempre combina bem com HTML que não está perfeitamente formado. Por esse motivo, usaremos um analisador diferente, que precisamos instalar:

In [25]:
```
!pip3 install html5lib
!pip3 install beautifulsoup4
!pip3 install requests
```

```
Collecting html5lib
  Downloading html5lib-1.1-py2.py3-none-any.whl (112 kB)
     |████████████████████████████████| 112 kB 8.3 MB/s eta 0:00:01
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.7/
site-packages (from html5lib) (1.15.0)
Requirement already satisfied: webencodings in /usr/local/lib/python
3.7/site-packages (from html5lib) (0.5.1)
Installing collected packages: html5lib
Successfully installed html5lib-1.1
Collecting beautifulsoup4
  Downloading beautifulsoup4-4.9.3-py3-none-any.whl (115 kB)
     |████████████████████████████████| 115 kB 6.7 MB/s eta 0:00:01
Collecting soupsieve>1.2; python_version >= "3.0"
  Downloading soupsieve-2.2.1-py3-none-any.whl (33 kB)
Installing collected packages: soupsieve, beautifulsoup4
Successfully installed beautifulsoup4-4.9.3 soupsieve-2.2.1
Requirement already satisfied: requests in /usr/local/lib/python3.7/
site-packages (2.25.0)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/p
ython3.7/site-packages (from requests) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python
3.7/site-packages (from requests) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/
python3.7/site-packages (from requests) (2020.6.20)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/l
ib/python3.7/site-packages (from requests) (1.26.2)
```

Para usar o Beautiful Soup, precisamos passar um pouco de HTML para a função BeautifulSoup (). Em nossos exemplos, isso será o resultado de uma chamada para requests.get:

In [26]:
```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

Depois disso, podemos ir muito longe usando alguns métodos simples. Normalmente, trabalhamos com os objetos Tag, que correspondem às tags que representam a estrutura de uma página HTML. Por exemplo, para encontrar a primeira tag

(e seu conteúdo), você pode usar:

In [27]:
```
first_paragraph = soup.find('p') # or just soup.p
```

In [28]:
```
print(first_paragraph)
```

```
<p>This domain is for use in illustrative examples in documents. You
may use this
    domain in literature without prior coordination or asking for pe
rmission.</p>
```

Você pode obter o conteúdo de texto de um Tag usando sua propriedade text:

In [29]:
```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

In [30]:
```
first_paragraph_text
```

Out[30]:
```
'This domain is for use in illustrative examples in documents. You m
ay use this\n    domain in literature without prior coordination or
asking for permission.'
```

In [31]:
```
first_paragraph_words
```

Out[31]:
```
['This',
 'domain',
 'is',
 'for',
 'use',
 'in',
 'illustrative',
 'examples',
 'in',
 'documents.',
 'You',
 'may',
 'use',
 'this',
 'domain',
 'in',
 'literature',
 'without',
 'prior',
 'coordination',
 'or',
 'asking',
 'for',
 'permission.']
```

E você pode extrair os atributos de uma tag tratando-a como um dict:

In [32]:
```
first_paragraph_id = soup.p['id'] # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id') # returns None if no 'id'
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent cal
l last)
<ipython-input-32-a2e731f68775> in <module>
----> 1 first_paragraph_id = soup.p['id'] # raises KeyError if no 'i
d'
      2 first_paragraph_id2 = soup.p.get('id') # returns None if no
'id'

/usr/local/lib/python3.7/site-packages/bs4/element.py in __getitem__
(self, key)
   1404         """tag[key] returns the value of the 'key' attribute
for the Tag,
   1405         and throws an exception if it's not there."""
-> 1406         return self.attrs[key]
   1407
   1408     def __iter__(self):

KeyError: 'id'
```

Você pode obter várias tags de uma só vez:

In [33]:
```
all_paragraphs = soup.find_all('p') # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

In [34]:
```
print(all_paragraphs)
```

```
[<p>This domain is for use in illustrative examples in documents. Yo
u may use this
    domain in literature without prior coordination or asking for pe
rmission.</p>, <p><a href="https://www.iana.org/domains/example">Mor
e information...</a></p>]
```

In [35]:
```
paragraphs_with_ids
```

Out[35]:
```
[]
```

Freqüentemente você vai querer encontrar tags com uma classe específica:

In [30]:
```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

E você pode combiná-los para implementar uma lógica mais elaborada. Por exemplo, se você quiser encontrar todos os elementos < span> contidos em um elemento < div>, poderá fazer isso:

In [31]:
```
# warning, will return the same span multiple times
# if it sits inside multiple divs
# be more clever if that's the case
spans_inside_divs = [span
                     for div in soup('div') # for each <div> on the
page
                     for span in div('span')] # find each <span> ins
ide it
```

Apenas esse punhado de recursos nos permitirá fazer bastante. Se você precisar fazer coisas mais complicadas (ou se estiver curioso), verifique a documentação.

É claro que qualquer informação importante não será rotulada como class = "important". Você precisará inspecionar cuidadosamente o HTML de origem, analisar sua lógica de seleção e preocupar-se com os casos de borda para garantir que seus dados estejam corretos. Vamos ver um exemplo.

Exemplos

O objetivo é examinar quantos livros de dados O'Reilly publicou ao longo do tempo. Depois de vasculhar seu site, você descobre que possui muitas páginas de livros de dados (e vídeos), acessíveis por meio de páginas de diretórios composto de 30 itens por vez com URLs como:

http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page=1 (http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicationDate&page=1)

A menos que você queira ser um idiota (e a menos que você queira que seu scraper seja banido), sempre que quiser extrair dados de um site, você deve primeiro verificar se ele tem algum tipo de política de acesso. Olhando para:

http://oreilly.com/terms/ (http://oreilly.com/terms/)

Parece não haver nada que proíba esse projeto. Para sermos bons cidadãos, devemos também procurar um arquivo robots.txt que diga ao webcrawlers como se comportar.

https://developers.google.com/search/docs/advanced/robots/create-robots-txt (https://developers.google.com/search/docs/advanced/robots/create-robots-txt)

As linhas abaixo são importantes em robots.txt são:

In [ ]:
```
Crawl-delay: 30
Request-rate: 1/30
```

O primeiro nos diz que devemos **esperar 30 segundos** entre os pedidos, o segundo que devemos solicitar **apenas uma página** a cada 30 segundos.
Então, basicamente, são duas maneiras diferentes de dizer a mesma coisa (há outras linhas que indicam que os diretórios não devem ser copiados, mas não incluem nosso URL, por isso, estamos bem lá.)

Um bom primeiro passo é encontrar todos os elementos da tag td thumbtext:

In [38]:
```
tds = soup('td', 'thumbtext')
print(len(tds)) # 30
```

```
0
```

Google - Capturar TAG IMG

In [104]:
```
from bs4 import BeautifulSoup
import requests

html = requests.get("http://www.google.com").text
soup = BeautifulSoup(html, 'html5lib')


tag_logo = soup.find_all('img')
tag_logo = [p for p in soup('img') if p.get('id') == 'hplogo']

print(tag_logo)
```

```
[<img alt="Google" height="92" id="hplogo" src="/images/branding/goo
glelogo/1x/googlelogo_white_background_color_272x92dp.png" style="pa
dding:28px 0 14px" width="272"/>]
```

In [ ]:
```

```

In [105]:
```
from bs4 import BeautifulSoup
import requests

html = requests.get("http://www.google.com").text
soup = BeautifulSoup(html, 'html5lib')

tag_logo = soup.find(id='hplogo')
print('\n', tag_logo, '\n')
print('Atributo SRC=', tag_logo.get('src'))
```

```
 <img alt="Google" height="92" id="hplogo" src="/images/branding/goo
glelogo/1x/googlelogo_white_background_color_272x92dp.png" style="pa
dding:28px 0 14px" width="272"/>

Atributo SRC= /images/branding/googlelogo/1x/googlelogo_white_backgr
ound_color_272x92dp.png
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: