Chapter 9: Getting the data

Prepared by,

Mrs.Ashritha K P

Asst. Professor

Dept. of ISE.

Sahyadri College of Engineering and Management

Introduction

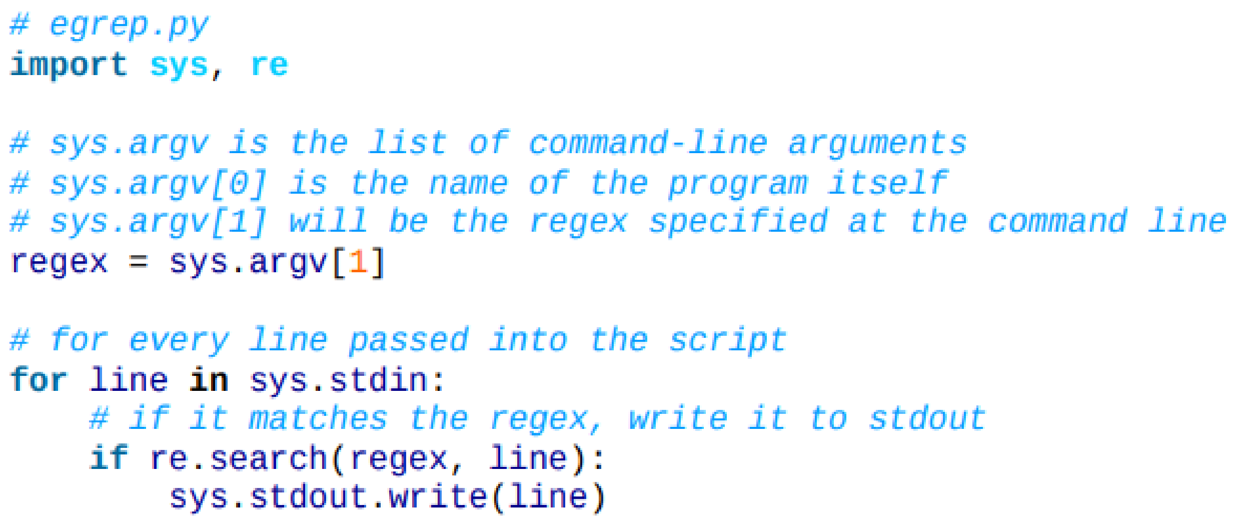
- In order to be a data scientist you need data.
- Data scientist will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data.
- Can always type the data in yourself but usually this is not a good use of your time.
- Here we are looking at different ways of getting data into Python and into the right formats

stdin, stdout

- Python’s sys module provides us with all three file objects for stdin, stdout, and stderr.
- Standard input – This is the file-handle that a user program reads to get information from the user. We give input to the standard input (stdin).
- The user program writes normal information to this file-handle. The output is returned via the Standard output (stdout).
- The user program writes error information to this file-handle. Errors are returned via the Standard error (stderr).

Here is a script that reads in lines of text and spits back out the ones that match a regular expression

- sys.argv is a list in Python that contains all the command-line arguments passed to the script.
- re.search() checks for a match anywhere in the string

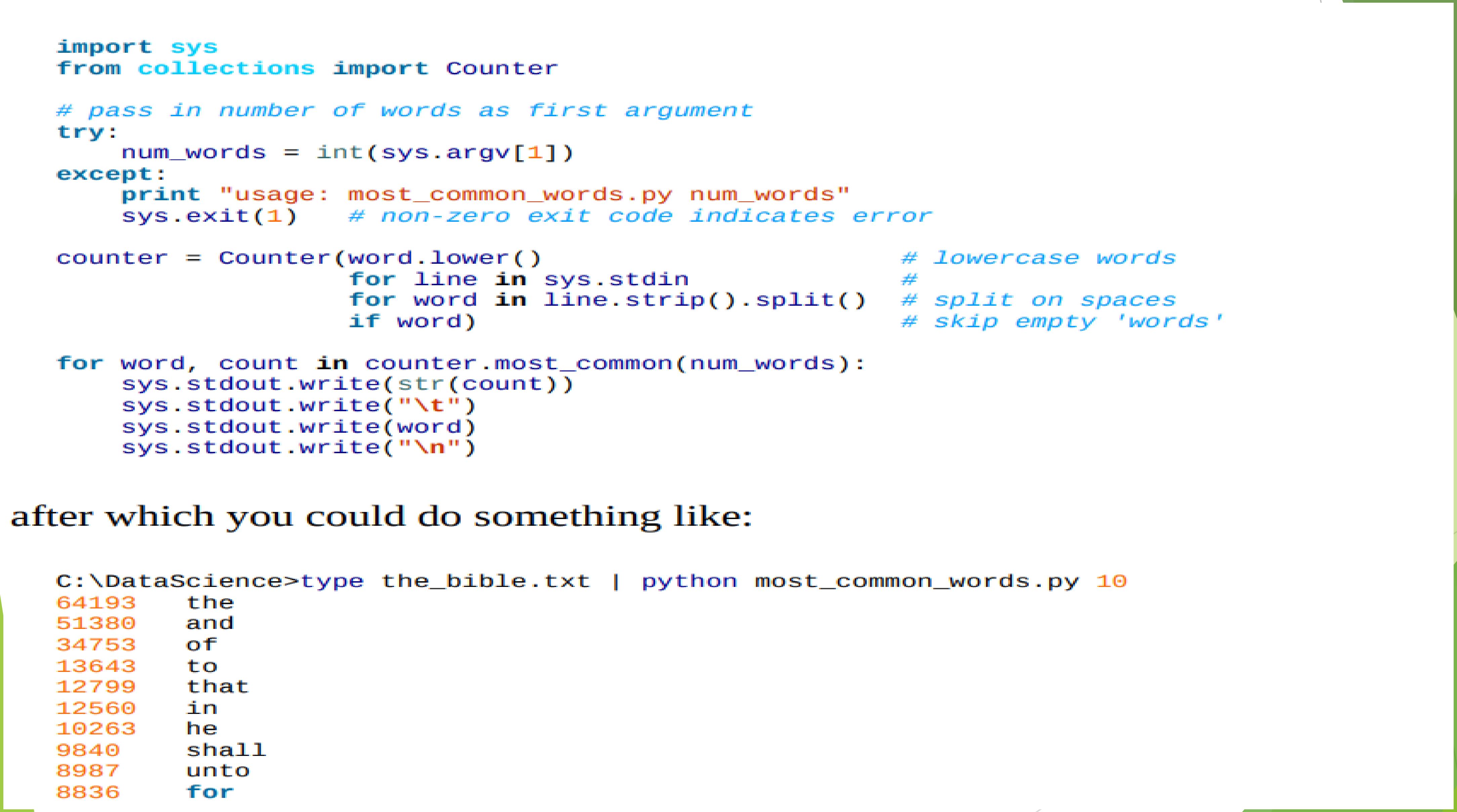


```
# egrep.py
import sys, re

# 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)
```

And here’s one that counts the lines it receives and then writes out the count:

```python
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

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

A script that counts the words in its input and writes out the most common ones: # most_common_words.



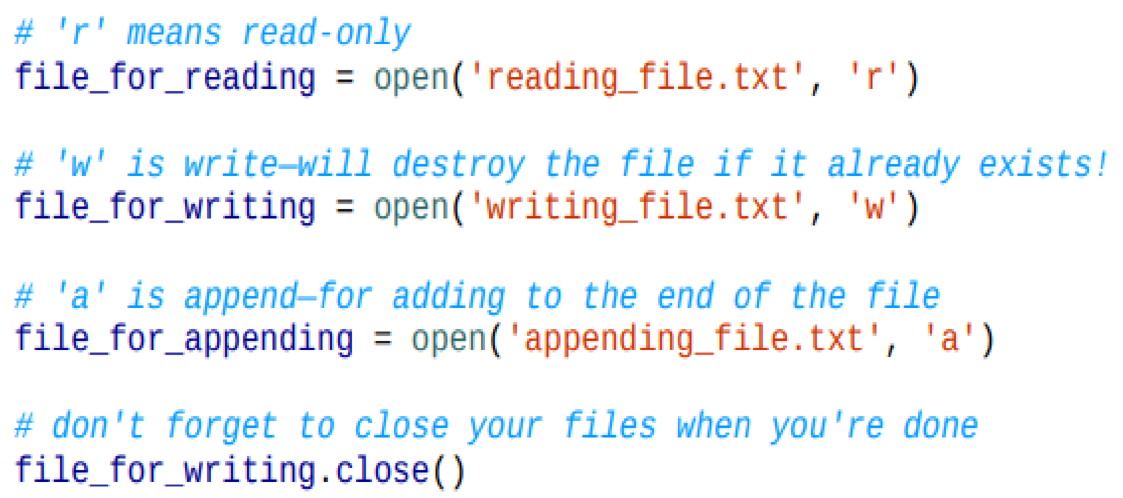
```
import sys
from collections import Counter

# 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 spaces
                  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:

```
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
```

Reading Files

- You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple
- **The Basics of Text Files:** The first step to working with a text file is to obtain a file object using open:



```python
# '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()
```

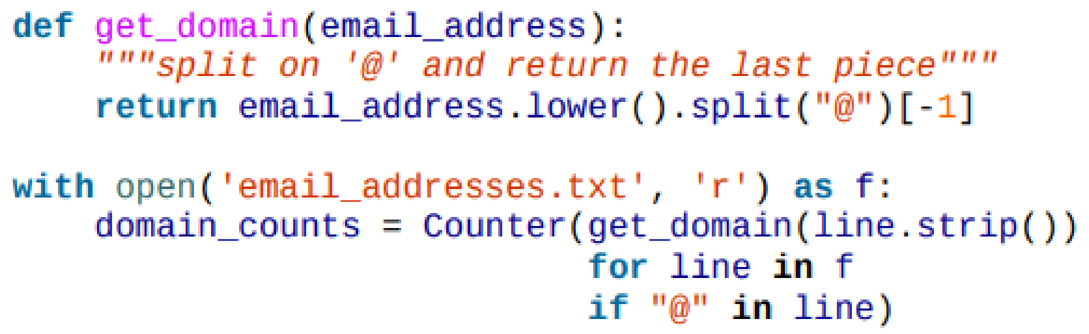
- Because it is easy to forget to close your files, you should always use them in a with block, at the end of which they will be closed automatically:

```
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)
```

- If you need to read a whole text file, you can just iterate over the lines of the file using for:

```
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 with '#'
            starts_with_hash += 1  # if it does, add 1 to the count
```

- Python strip() function is used to remove extra whitespaces and specified characters from the start and from the end of the strip irrespective of how the parameter is passed.
- You have a file full of email addresses, one per line, and that you need to generate a histogram of the domains. A good first approximation is to just take the parts of the email addresses that come after the @.



```
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)
```

Delimited Files

- More frequently we'll work with files with lots of data on each line.
- These files are very often either comma-separated or tab-separated.
- Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.
- We use Python's csv module (or the pandas library)
- we should always work with csv files in binary mode by including a b after the r or w
- We can use csv.reader to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
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
```

we could process them with:

```
import csv

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

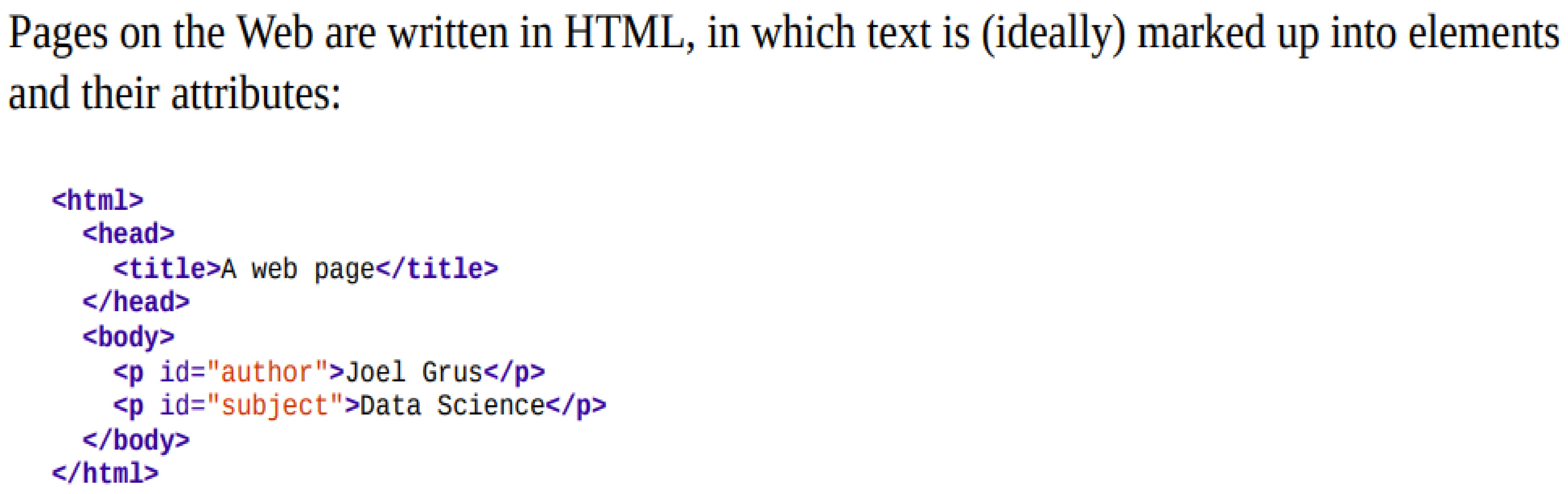
If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

- get each row as a dict (with the headers as keys) by using csv.DictReader:

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

HTML and the Parsing Thereof



Pages on the Web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

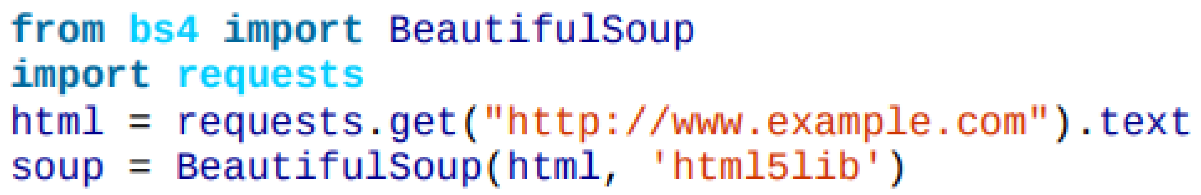
```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

- To get data out of HTML, we will use the **BeautifulSoup** library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them.

- We use requests library (pip install requests), which is a much nicer way of making HTTP requests ,that's built into Python
- Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed.
- so, we'll use a different parser, which we need to install

  pip install html5lib

- To use Beautiful Soup, we'll need to pass some HTML into the BeautifulSoup() function.



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

- We’ll typically work with Tag objects, which correspond to the tags representing the structure of an HTML page.
- For example, to find the first <p> tag (and its contents) you can use:

  first_paragraph = soup.find('p’)

- You can get the text contents of a Tag using its text property

  first_paragraph_text = soup.p.text

  first_paragraph_words = soup.p.text.split()

- And you can extract a tag’s attributes by treating it like a dict:

  first_paragraph_id = soup.p['id’]

  first_paragraph_id2 = soup.p.get('id’)

- You can get multiple tags at once:

  all_paragraphs = soup.find_all('p’)

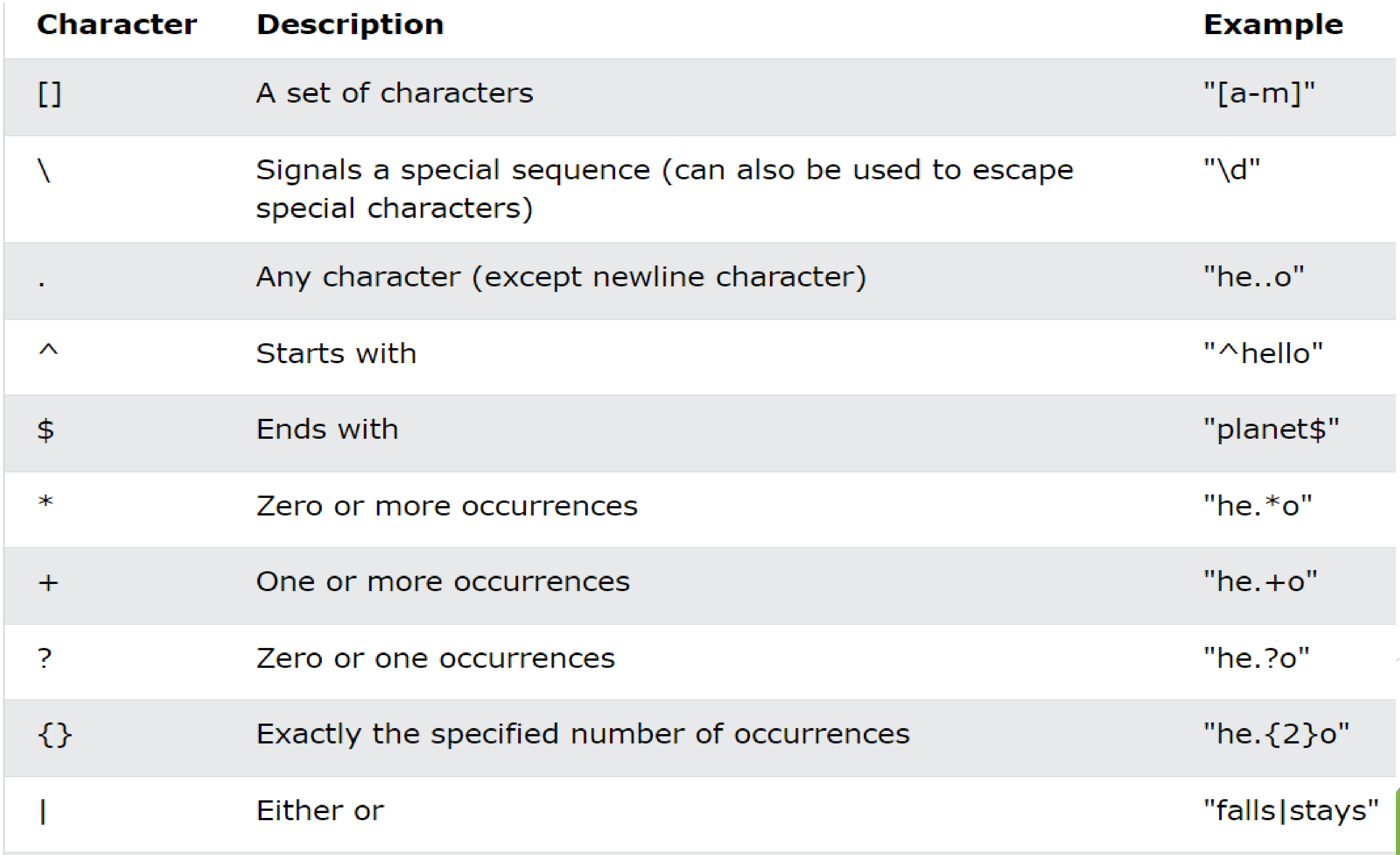
  paragraphs_with_ids = [p for p in soup('p') if p.get('id')]

```
# html code
html_doc = """<html><head><title>Welcome  to
geeksforgeeks</title></head>
<body>
<p class="title"><b>Geeks</b></p>


<p class="body">geeksforgeeks a computer science portal for geeks
</body>
"""

# import module
from bs4 import BeautifulSoup

# parse html content
soup = BeautifulSoup( html_doc , 'html.parser')

# Finding by class name
soup.find( class_ = "body" )
```

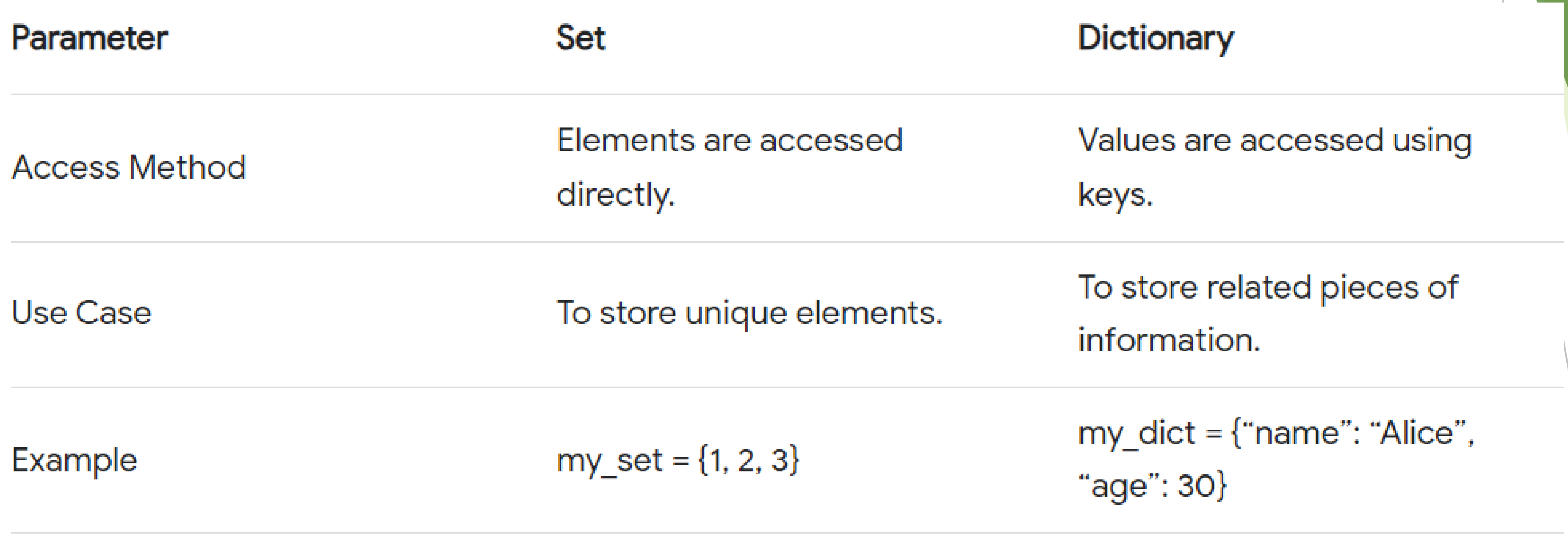
Output:

```
<p class="body">geeksforgeeks a computer science portal for geeks
</p>
```

regex = r"^https?://.*\.house\.gov/?$"



Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"



Parameter	Set	Dictionary
Access Method	Elements are accessed directly.	Values are accessed using keys.
Use Case	To store unique elements.	To store related pieces of information.
Example	my_set = {1, 2, 3}	my_dict = {“name”: “Alice”, “age”: 30}

- There are different ways to pass values to a dictionary.
- One way is to pass values into the dict() constructor is to provide the key-value pairs as a list of tuples.
- `my_dictionary = dict([('key1', 'value1'), ('key2', 'value2'), ('key3', 'value3')])`

- F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.
- val = ‘Hello'
- print(f"{val}to greet. {val} is way to greet.say {val}.")
- **o/p**
- Hello to greet. Hello is way to greet.say Hello.
-
- name = ‘john'
- age = 23
- print(f"Hello, My name is {name} and I'm {age} years old.")

Using API’s

- APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.
- For example, the weather bureau’s software system contains daily weather data. The weather app on your phone “talks” to this system via APIs and shows you daily weather updates on your phone.
- HTTP is protocol for transferring text. Data that we request through a web API need to be serialized into a string format.
- JavaScript Object Notation (JSON) is the data exchange format that makes this possible.
- JSON has become popular as a data format for developers because of its human-readable text, which is lightweight, requires less coding, and processes faster since deserialization isn't required.

- Parse JSON using Pythons json module.
- loads() function deserializes a string representing a JSON object into a Python object.
- import json
- serialized = """{ "title" : "Data Science Book",
-                   "author": "Joel Grus",
-                   "publicationYear": 2019,
-                   "topics" : ["data", "science", "data science"]}"""
- deserialized = json.loads(serialized)
- assert deserialized["publicationYear"] == 2019
- assert "data science" in deserialized["topics"]

Using an Unauthenticated API

- Most APIs these days require you to first authenticate yourself in order to use them.
- first take a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"
repos = json.loads(requests.get(endpoint).text)
```

- Date fromat is string
  "created_at": u'2013-07-05T02:02:28Z'
- So we need dateutil parser

- Because HTTP is a protocol for transferring text, the data you request through a web API
- needs to be serialized into a string format. Often this serialization uses JavaScript Object
- Notation (JSON). JavaScript objects look quite similar to Python dicts, which makes their
- string representations easy to interpret:
- { "title" : "Data Science Book",
- "author" : "Joel Grus",
- "publicationYear" : 2014,
- "topics" : ["data", "science", "data science"] }

Example: Using the Twitter APIs

- To interact with the Twitter APIs we'll be using the Twython library (pip install twython).
- In order to use Twitter's APIs, you need to get some credentials (for which you need a Twitter account)

1. go to https://developer.twitter.com/
2. If you are not signed in, click Sign in and enter your Twitter username and password.
3. Click Apply to apply for developer account
4. Request access for your own personal use.
5. Fill in the application, it frequires 300 words on why you need to access.
6. Wait for indefinite amount of time.
7. Once you get approved go back to developer.twitter.com. Find "apps" section. And click "create an app"
8. Fill all the required fields.
9. Click create.