Starting to Crawl

So far, the examples in the book have covered single static pages, with somewhat artificial canned examples. In this chapter, we'll start looking at some real-world problems, with scrapers traversing **multiple pages and even multiple sites**.
Web crawlers are called such because they crawl across the Web. **At their core is an element of recursion.** They must retrieve page contents for a **URL**, examine that page for another URL, and retrieve that page, ad infinitum.
Beware, however: just because you can crawl the Web doesn't mean that you always should. The scrapers used in previous examples work great in situations where all the data you need is on a single page. With web crawlers, you must be extremely conscientious of how much bandwidth you are using and make every effort to determine if there's a way to make the target server's load easier.

Traversing a Single Domain

In this section, we'll begin a project that will become a "Six Degrees of Wikipedia" solution finder. That is, we'll be able to take the Eric Idle page and find the fewest number of link clicks that will take us to the Kevin Bacon page.

You should already know how to write a Python script that retrieves an arbitrary Wikipedia page and produces a list of links on that page:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
html = urlopen("http://en.wikipedia.org/wiki/Kevin_Bacon")
bsObj = BeautifulSoup(html)
for link in bsObj.findAll("a"):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

If you look at the list of links produced, you'll notice that all the articles you'd expect are there: "Apollo 13," "Philadelphia," "Primetime Emmy Award," and so on. However, there are some things that we don't want as well:

In fact, Wikipedia is full of sidebar, footer, and header links that appear on every page, along with links to the category pages, talk pages, and other pages that do not contain different articles:

If you examine the links
that point to article pages (as opposed to other internal pages), they all have three things in
Common:

1.They reside within the div with the id set to bodyContent
2.The URLs do not contain semicolons
3.The URLs begin with /wiki/

We can use these rules to revise the code slightly to retrieve only the desired article links:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen("http://en.wikipedia.org/wiki/Kevin_Bacon")
bsObj = BeautifulSoup(html)
for link in bsObj.find("div", {"id":"bodyContent"}).findAll("a",href=re.compile("^(/wiki/)((?!:).)*$")):
   if 'href' in link.attrs:
      print(link.attrs['href'])
```

If you run this, you should see a list of all article URLs that the Wikipedia article on Kevin Bacon links to.
Of course, having a script that finds all article links in one, hardcoded Wikipedia article, while interesting, is fairly useless in practice. We need to be able to take this code and transform it into something more like the following:

A single function, getLinks, that takes in a Wikipedia article URL of the form /wiki/<Article_Name> and returns a list of all linked article URLs in the same form.

A main function that calls getLinks with some starting article, chooses a random article link from the returned list, and calls getLinks again, until we stop the program or until there are no article links found on the new page.

Here is the complete code that accomplishes this:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re
random.seed(datetime.datetime.now())

def getLinks(articleUrl):
    html = urlopen("http://en.wikipedia.org"+articleUrl)
    bsObj = BeautifulSoup(html)
    return bsObj.find("div", {"id":"bodyContent"}).findAll("a",href=re.compile("^(/wiki/)((?!:).)*$"))

links = getLinks("/wiki/Kevin_Bacon")
while len(links) > 0:
    newArticle = links[random.randint(0, len(links)-1)].attrs["href"]
    print(newArticle)
    links = getLinks(newArticle)
```

The first thing the program does, after importing the needed libraries, is set the random number generator seed with the current system time. This practically ensures a new and interesting random path through Wikipedia articles every time the program is run.

Next, it defines the getLinks function, which takes in an article URL of the form /wiki/..., prepends the Wikipedia domain name, http://en.wikipedia.org, and retrieves the BeautifulSoup object for the HTML at that domain. It then extracts a list of article link tags, based on the parameters discussed previously, and returns them.
The main body of the program begins with setting a list of article link tags (the links variable) to the list of links in the initial page: http://bit.ly/1KwqjU7. It then goes into a loop, finding a random article link tag in the page, extracting the href attribute from it, printing the page, and getting a new list of links from the extracted URL.
Of course, there's a bit more to solving a "Six Degrees of Wikipedia" problem than simply building a scraper that goes from page to page. We must also be able to store and analyze the resulting data. For a continuation of the solution to this problem, see **Chapter 5.**

Crawling an Entire Site

In the previous section, we took a random walk through a website, going from link to link. But what if you need to systematically catalog or search every page on a site? Crawling an entire site, especially a large one, is a memory-intensive process that is best suited to applications where a database to store crawling results is readily available. However, we can explore the behavior of these types of applications without actually running them fullscale. To learn more about running these applications using a database, see Chapter 5.

So when might crawling an entire website be useful and when might it actually be harmful? Web scrapers that traverse an entire site are good for many things, including:

Generating a site map.
A few years ago, I was faced with a problem: an important client wanted an estimate for a website redesign, but did not want to provide my company with access to the internals of their current content management system, and did not have a publicly available site map. I was able to use a crawler to cover their entire site, gather all internal links, and organize the pages into the actual folder structure they had on their site. This allowed me to quickly find sections of the site I wasn't even aware existed, and accurately count how many page designs would be required, and how much content would need to be migrated.

Gathering data
Another client of mine wanted to gather articles (stories, blog posts, news articles, etc.) in order to create a working prototype of a specialized search platform. Although these website crawls didn't need to be exhaustive, they did need to be fairly expansive (there were only a few sites we were interested in getting data from). I was able to create crawlers that recursively traversed each site and collected only data it found on article pages.

The general approach to an exhaustive site crawl is to start with a top-level page (such as the home page), and search for a list of all internal links on that page. Every one of those links is then crawled, and additional lists of links are found on each one of them, triggering another round of crawling.

Clearly this is a situation that can blow up very quickly. If every page has 10 internal links, and a website is five pages deep (a fairly typical depth for a medium-size website), then the number of pages you need to crawl is , or 100,000 pages, before you can be sure that you've exhaustively covered the website. Strangely enough, while "5 pages deep and 10 internal links per page" are fairly typical dimensions for a website, there are very few websites with 100,000 or more pages. The reason, of course, is that the vast majority of internal links are duplicates.

In order to avoid crawling the same page twice, it is extremely important that all internal links discovered are formatted consistently, and kept in a running list for easy lookups, while the program is running. Only links that are “new” should be crawled and searched for additional links:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen("http://en.wikipedia.org"+pageUrl)
    bsObj = BeautifulSoup(html)
    for link in bsObj.findAll("a", href=re.compile("^(/wiki/)")):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                #We have encountered a new page
                newPage = link.attrs['href']
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks("")
```

In order to get the full effect of how this web-crawling business works, I’ve relaxed the standards of what constitutes an “internal link we are looking for,” from previous examples. Rather than limit the scraper to article pages, it looks for all links that begin with /wiki/ regardless of where they are on the page, and regardless of whether they contain colons. Remember: article pages do not contain colons, but file upload pages, talk pages, and the like do contain colons in the URL).

Initially, getLinks is called with an empty URL. This is translated as “the front page of Wikipedia” as soon as the empty URL is prepended with http://en.wikipedia.org inside the function. Then, each link on the first page is iterated through and a check is made to see if it is in the global set of pages (a set of pages that the script has encountered already). If not, it is added to the list, printed to the screen, and the getLinks function is called recursively on it.

Collecting Data Across an Entire Site

Of course, web crawlers would be fairly boring if all they did was hop from one page to the other. In order to make them useful, we need to be able to do something on the page while we're there. Let's look at how to build a scraper that collects the title, the first paragraph of content, and the link to edit the page (if available).

As always, the first step to determine how best to do this is to look at a few pages from the site and determine a pattern. By looking at a handful of Wikipedia pages both articles and non-article pages such as the privacy policy page, the following things should be clear:

All titles (on all pages, regardless of their status as an article page, an edit history page, or any other page) have titles under h1→span tags, and these are the only h1 tags on the page.

As mentioned before, all body text lives under the div#bodyContent tag. However, if we want to get more specific and access just the first paragraph of text, we might be better off using div#mw-content-text → p (selecting the first paragraph tag only). This is true for all content pages except file pages (for example: http://bit.ly/1KwqJtE), which do not have sections of content text.

Edit links occur only on article pages. If they occur, they will be found in the li#caedit tag, under li#ca-edit → span → a.

By modifying our basic crawling code, we can create a combination crawler/datagathering (or, at least, data printing) program:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
import re
pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen("http://en.wikipedia.org"+pageUrl)
    bsObj = BeautifulSoup(html)
    try:
        print(bsObj.h1.get_text())
        print(bsObj.find(id ="mw-content-text").findAll("p")[0])
        print(bsObj.find(id="ca-edit").find("span").find("a").attrs['href'])
    except AttributeError:
        print("This page is missing something! No worries though!")
    for link in bsObj.findAll("a", href=re.compile("^(/wiki/)")):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                #We have encountered a new page
                newPage = link.attrs['href']
                print("----------------\n"+newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks("")
```

The for loop in this program is essentially the same as it was in the original crawling program (with the addition of some printed dashes for clarity, separating the printed content).

Because we can never be entirely sure that all the data is on each page, each print statement is arranged in the order that it is likeliest to appear on the site. That is, the <h1> title tag appears on every page (as far as I can tell, at any rate) so we attempt to get that data first. The text content appears on most pages (except for file pages), so that is the second piece of data retrieved. The “edit” button only appears on pages where both titles and text content already exist, but it does not appear on all of those pages.

You might notice that in this and all the previous examples, we haven’t been “collecting” data so much as “printing” it. Obviously, data in your terminal is hard to manipulate. We’ll look more at storing information and creating databases in Chapter 5.

Crawling Across the Internet

Before you start writing a crawler that simply follows all outbound links willy nilly, you should ask yourself a few questions:

What data am I trying to gather? Can this be accomplished by scraping just a few predefined websites (almost always the easier option), or does my crawler need to be able to discover new websites I might not know about?

When my crawler reaches a particular website, will it immediately follow the next outbound link to a new website, or will it stick around for a while and drill down into the current website?

Are there any conditions under which I would not want to scrape a particular site? Am I interested in non-English content?

How am I protecting myself against legal action if my web crawler catches the attention of a webmaster on one of the sites it runs across? (Check out Appendix C for more information on this subject.)

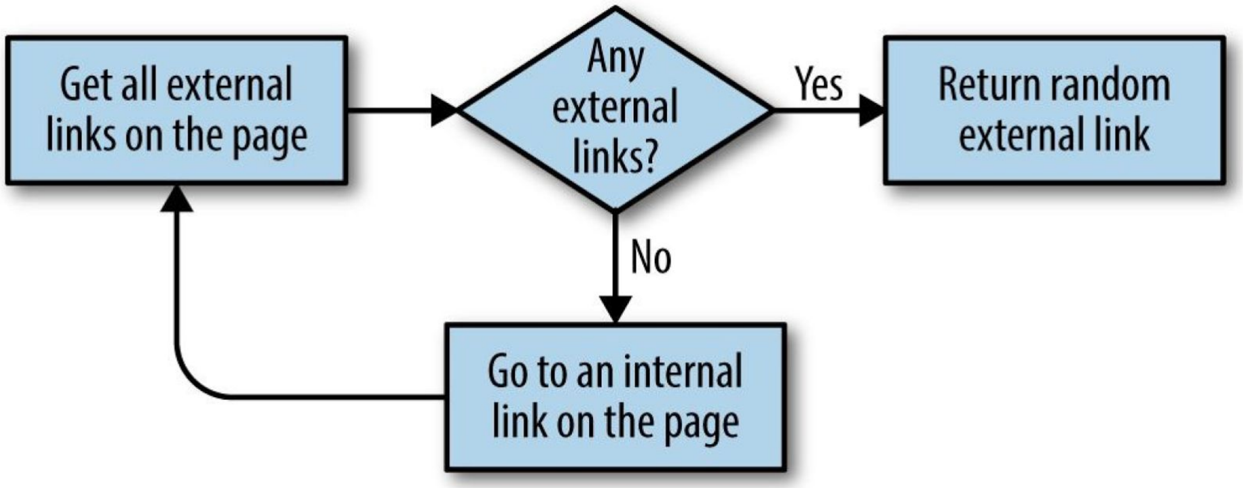
A flexible set of Python functions that can be combined to perform a variety of different types of web scraping can be easily written in fewer than 50 lines of code:

```
from urllib2 import urlopen
from bs4 import BeautifulSoup
import re
import datetime
import random
pages = set()
random.seed(datetime.datetime.now())
#Retrieves a list of all Internal links found on a page
def getInternalLinks(bsObj, includeUrl):
    internalLinks = []
    #Finds all links that begin with a "/"
    for link in bsObj.findAll("a", href=re.compile("^(/|.*"+includeUrl+")")):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in internalLinks:
                internalLinks.append(link.attrs['href'])
    return internalLinks

#Retrieves a list of all external links found on a page
def getExternalLinks(bsObj, excludeUrl):
    externalLinks = []
    #Finds all links that start with "http" or "www" that do
    #not contain the current URL
    for link in bsObj.findAll("a",href=re.compile("^(http|www)((?!"+excludeUrl+").)*$")):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in externalLinks:
                externalLinks.append(link.attrs['href'])
    return externalLinks
def splitAddress(address):
    addressParts = address.replace("http://", "").split("/")
    return addressParts
def getRandomExternalLink(startingPage):
    html = urlopen(startingPage)
    bsObj = BeautifulSoup(html)
    externalLinks = getExternalLinks(bsObj, splitAddress(startingPage)[0])
    if len(externalLinks) == 0:
        internalLinks = getInternalLinks(startingPage)
        return getNextExternalLink(internalLinks[random.randint(0,len(internalLinks)-1)])
    else:
        return externalLinks[random.randint(0, len(externalLinks)-1)]
```

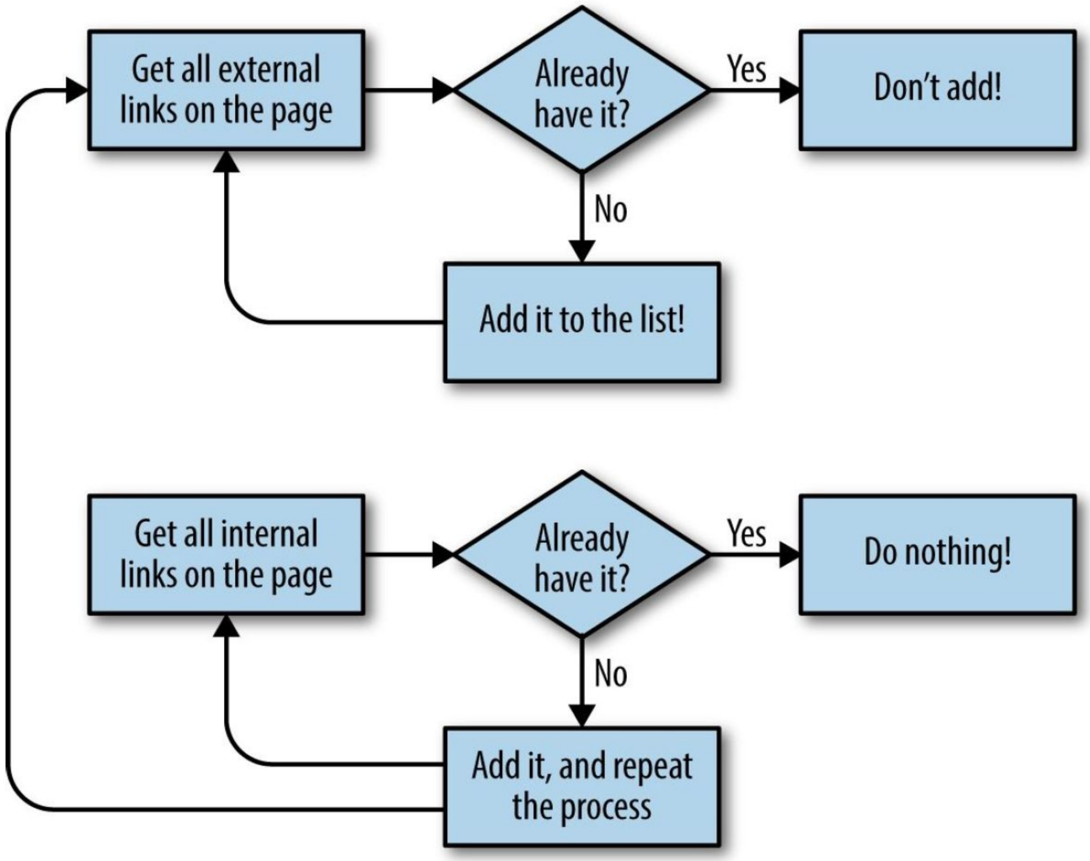
```
def followExternalOnly(startingSite):
    externalLink = getRandomExternalLink("http://oreilly.com")
    print("Random external link is: "+externalLink)
    followExternalOnly(externalLink)
followExternalOnly("http://oreilly.com")
```

The nice thing about breaking up tasks into simple functions such as “find all external links on this page” is that the code can later be easily refactored to perform a different crawling task. For example, if our goal is to crawl an entire site for external links, and make a note of each one, we can add the following function:


Get all external links on the page
Any external links?
Yes
Return random external link
No
Go to an internal link on the page

```
allExtLinks = set()
allIntLinks = set()
def getAllExternalLinks(siteUrl):
    html = urlopen(siteUrl)
    bsObj = BeautifulSoup(html)
    internalLinks = getInternalLinks(bsObj,splitAddress(siteUrl)[0])
    externalLinks = getExternalLinks(bsObj,splitAddress(siteUrl)[0])
    for link in externalLinks:
        if link not in allExtLinks:
            allExtLinks.add(link)
            print(link)
    for link in internalLinks:
        if link not in allIntLinks:
            print("About to get link: "+link)
            allIntLinks.add(link)
            getAllExternalLinks(link)
getAllExternalLinks("http://oreilly.com")
```

This code can be thought of as two loops — one gathering internal links, one gathering external links — working in conjunction with each other. The flowchart looks something like Figure 3-2:


Get all external links on the page
Already have it?
Yes
Don't add!
No
Add it to the list!
Get all internal links on the page
Already have it?
Yes
Do nothing!
No
Add it, and repeat the process