INTRODUCTION TO PROGRAMMING
WEEK 9 - WEB CRAWLING AND BASIC REGEX

FINAL EXAM

Your final exam is coming up in 2 short weeks! The two part exam will take place on 3/23.

To pass this class, you will need to:

- Get a grade of B or better on final exam
- Complete ALL homework assignments before 12:00AM PST on 3/29

If you are behind on homework, I would strongly recommend catching up now instead of waiting until the last minute, since you will also need time to study for the exam.

GETTING THE DEMO FILES

This week's class will move very fast, so I have prepared demo files for you to follow along with.

Please run the following from your week9 directory:

```
cp -r /home/alberthwang/python/week9 demo_files
```

All the examples below reference something in the starter files. Please note that these starter files may change, but I will endeavor to keep them as up to date as possible. To refresh your version, just run:

```
rm -r demo_files
cp -r /home/alberthwang/python/week9 demo_files
```

REGULAR EXPRESSIONS

Regular expressions are expressions that delineate string patterns to be matched. Here is an example - imagine if we had to process 100 forms that all looked like this:

Name: Albert Hwang
Location: US-MTV-CL5
Department: People Operations

How would we extract the information in the form? To us it seems obvious, but how do you tell a computer to "get the information after 'Name:' but before 'Location:, it will be 2 words separated by 1 space"?

Characters

While regular expressions, or regex, can get very complex very quickly, we will only be looking at some basic patterns in this class - enough to get you started. In regex, there are a few characters that specify patterns.

```
my_str = 'this is the song that never ends. It goes on 2day, 2morrow...'
```

`\w`	represents any letter in the alphabet. In `my_str` this would match `['t', 'h', 'i', 's', 'i',...]`
`\s`	represents white space of any kind including spaces(' '), tabs (`\t`), carriages (`\r`), and newlines (`\n`). In `my_str` this would be spaces and newlines - `[' ', ' ',...]`
`\d`	represents digits, or numerical units. In `my_str` this would be `[2,2]`
`.`	represents any character that's not a new line. In `my_str` this would be all the characters, digits, and punctuation marks.

There are several more characters used in regex, but these are the basic ones we will go over and use in this class.

Patterns

Patterns are sets of characters put together to help the script find matches in the string. Take a look at the following patterns and the matches they find:

```
my_str = 'this is the song that never ends. It goes on 2day, 2morrow...'
```

`\w\w\w\w`	`'this', 'song', 'that', 'ends', 'goes'`
`\d\w\w\w`	`'2day', '2mor'`
`\s\w\w\s`	`' is ', ' It ', ' on '`

There are even a few special matching characters that aid in finding patterns. These are the **special repetition operators**, they continue to search for a pattern until a match is not found.

`+`	This means 1 or more of the same character type or value
`*`	This means 0 or more of the same character type or value

Here are some examples:

`\w+`	`'this', 'is', 'the', 'song'`, etc. (matches all words)

\d*\w+	‘2day’, ‘2morrow’, ‘never’, etc.
\d+\w+	‘2day’, ‘2morrow’

Some common uses that you see of this are (.+) which will match anything up to a newline or (\w+) which will match any word (assuming a space or non-char follows the word). (\d+) matches integers.

A regular expression is essentially a combination of patterns, characters, and groups that is used to find combinations of substrings inside a string. This will all be clear with more examples.

Matching and Groups

To take full advantage of regex, you should combine context with matching patterns to pull out matching groups. Groups are sub-patterns within the matching pattern that are separated by parentheses. For example, let’s look at the forms from the introduction:

Name: Albert Hwang
Location: US-MTV-CL5
Department: People Operations

Name: Ryan Vukelich
Location: US-NYC-9th
Department: People Operations

Let’s look at some patterns and the groups they extract:

Name:(.+)	[‘ Albert Hwang’, ‘ Ryan Vukelich’]
Name:\s(\w+)\s(\w+)	[(‘Albert’, ‘Hwang’), (‘Ryan’, ‘Vukelich’)]
US-(\w+)-	[‘MTV’, ‘NYC’]

As you can see, we can use these matching patterns to extract things like full name, first name, last name, US office location, etc.

Greed Limiter

The “+” and “*” characters are greedy characters in regex in that they continue matching until no match is found. This greed can often have strange implications. For instance, what if I had a string like this?

```
Checkouts: start: 9/2, end: 10/23|start: 10/24, end: 12/2|
```

Let’s say I wanted to extract the start dates? The logical pattern I would choose is:

```
start:\s(.+), end
```

But since the “+” is greedy, it doesn’t stop at the first match and would return all of this in the matched group:

```
'9/2, end: 10/23|start: 10/24'
```

This is because it continues matching “.” chars until the last occurrence of “, end” (which occurs twice).

This solution for this is to use the “?” operator which limits the greed of the “+” and “*” characters to the first match.

```
'start:(.+?), end'
```

This will match up until the first occurrence of “, end”. This will become very important later when patterns repeat in the same string.

Special characters

As you can tell, there are several special characters here that are reserved for matching patterns such as **“\”**, **“+”, “.”, “*”, “?”** (and others like **“|”).** What if you want to match a pattern that contains one of these characters? Such as pulling the month, day, and year from this date:

```
09.02.2010
```

To add periods (or any special character) into a regex pattern, you need to escape it with a backslash. Here is the pattern above:

```
(\d+)\.(\d+)\.(\d+)
```

To designate the **“.”** character, I write **“\.”** putting a backslash in front to escape it.

Python Regex Methods

Now that you understand basic patterns, you can use python to extract these patterns. In order to work with regular expressions you need to first import the regular expressions module, `re`.

```
import re
```

The `re` module has a few very useful functions for extracting patterns. The function that is probably most useful in my opinion and that I use the most is **`findall`**.

FINDALL

The findall function takes 2 parameters, a regular expression and a string to match. It returns a list of matches. If no group is specified, then a list of strings with full matches is returned. If 1

group is specified, then a list of strings with the group matches are returned. If multiple groups are specified in the pattern, then a list of tuples (of strings) with the group matches positionally indexed from left to right is returned.Finally, if no matches are found, an empty list is returned. This will all make more sense with examples.

First, create a file called **employees.txt** and add the content from above with a little more:

```
Report Title: Employee Records
Last Updated: 3/9/2011

Name: Albert Hwang
Location: US-MTV-CL5
Department: People Operations

Name: Ryan Vukelich
Location: US-NYC-9th
Department: Sales
```

Next, create a Python file called **findall_demo.py** and add the following content:

```
import re #1

def main():
  employees_file = open('employees.txt', 'r')
  employees_str = employees_file.read() #2

  location_list = re.findall(r'Location:.+', employees_str) #3
  print location_list #4
  print ''

if __name__ == '__main__':
  main()
```

A few things to notice:

1. To work with regular expressions, we need to import the **`re`** module
2. Next, we need to get the file contents as a string
3. `findall` is a method of the `re` module and it takes 2 arguments - the pattern and the string to match it on.
4. `findall` returns a list object as its return type

```
re.findall(pattern, target_string)
```

The regular expression that `findall` takes is a raw string (the **r** in the front of the string tells us that) that describes the pattern we are trying to match. In our case, the pattern we're looking for is:

```
Location:.+
```

In this case we're saying, find every matching pattern that starts with the string "`Location:`" and is followed by **".+"**. From the notes, you may remember that the period, **"."**, can represent

any alphanumeric character except a new line. The **“+”** sign tells us that it can be 1 or more of **“.”** type characters. **Essentially this pattern is saying, “find me all patterns that begin with “**`Location:`**” followed by any number of characters of any kind up to a new line”.**

When we run the above example the output is:

```
['Location: US-MTV-CL5', 'Location: US-NYC-9th']
```

Groups

Of course, we’re often more interested in the actual location itself and not the text, “`Location:`”. For this, add parentheses around the **“.+”** to specify that we want just this group and not the whole matched pattern.

```
location_list = re.findall(r'Location:(.+)', employees_str)
```

Now when you print `location_list`, you get the following:

```
[' US-MTV-CL5', ' US-NYC-9th']
```

When 1 group is specified in the regular expression pattern, findall returns a list of strings which are the group matches.

How would we refine the regular expression even more so that we don’t get the space in front of the location?

```
location_list = re.findall(r'Location:\s(.+)', employees_str)
```

Multiple Groups

Often, we need to find multiple groups in a pattern. In our example, perhaps we want to extract the first name and last name of each employee. We know that this is located after the words “Name” and that there is a space separating “Name:” and the first name as well as a space separating the first name and the last name. So let’s get to work:

```
name_list = re.findall(r'Name:\s(\w+)\s(\w+)', employees_str)
```

Now when you print `name_list`, you get the following:

```
[('Albert', 'Hwang'), ('Ryan', 'Vukelich')]
```

When multiple groups are specified in a pattern, `findall` returns a list of tuples. Each tuple contains the groups inside the the matches in the order they are found, left-to-right.

Tuples are essentially immutable lists which means they are lists that cannot be changed. Since they behave like lists, you can loop through them and access them through indexes. For instance, if I wanted to print a list of the last names I could do the following:

```
# Print last names
for name in name_list:
  print name[1]
```

How would we extract the information for country, office, and building in the location string?

```
loc_exp = r'Location:\s(\w+)-(\w+)-(.+)'
location_list = re.findall(loc_exp, employees_str)
```

As you can see here, the more precise you make your match pattern, the easier it is to extract information out of pattern. Precision is the name of the game when it comes to regular expressions performance.

What if no matches are found?

If no matches are found, an empty list is returned from re.findall(). Of course you can't find elements in an empty list so running the following will throw an error (IndexError):

```
  # Testing if match found
  cc_match_list = re.findall(r'Cost Center:\s(\d+)',
                             employees_str)
  print cc_match_list[1]
```

The solution, of course, is to test using an if statement like so:

```
  # Testing if match found
  cc_match_list = re.findall(r'Cost Center:\s(\d+)',
                             employees_str)
  if len(cc_match_list) > 0:
    print cc_match_list
  else:
    print 'No Cost Centers Found.'
```

Demonstration of Greed Limiter

For a full demonstration of how to limit the greed of "+" and "*" you can take a look at these two files:

```
/home/alberthwang/python/week9/regex/greedy_demo.py
/home/alberthwang/python/week9/regex/library_data.txt
```

This concept will be important for the homework, so I recommend looking over these files.

SEARCH

The **`findall`** method will search an entire string from front to back searching for the pattern. Often, however, we know that there will only be 1 occurrence of the pattern, so we'd rather not waste the interpreter's time searching the entire string. In these cases, its better to use the **`search`** method which returns a **`MatchObject`** as its return type. This type of object has its own methods.

In our example, we know there is only 1 occurrence of the string “Report Title:”, so we’ll change the code in examples.py to this:

```
import re

def main():
  employees_file = open('employees.txt', 'r')
  employees_str = employees_file.read()

  title_match= re.search(r'Report Title:\s(.+)', employees_str) #1
  print title_match.group(0) #2 - Report Title: Employee Records
  print title_match.group(1) #3 - Employee Records
  print ''


if __name__ == '__main__':
  main()
```

Things to note here:

1. Here we are using the **`.search()`** method which has the same function signature as **`findall().`**
2. The search method returns a `MatchObject` that has a method called **`group().`** When the `group()` method is passed `0`, it returns the entire match.
3. When the `group()` method is passed a numerical index, it returns the group match in that order it was found, so `1` will return the 1st group match and `2` will return the 2nd.

Finding one value but with multiple groups - search()

Like findall(), search() also supports returning multiple groups. How would we extract the month, date, and year out of the “Last Updated” field in the employees files?

```
  # search for one match but with multiple groups
  updated_match = re.search(r'Last Updated:\s(\d+)/(\d+)/(\d+)',
                            employees_str)
  print updated_match.group(0) # Last Updated 3/9/2011
  print updated_match.group(1) # 3
  print updated_match.group(2) # 9
  print updated_match.group(3) # 2011
```

What if multiple matches are found?

If multiple matches to the pattern are found, then search will return the First matched pattern.

```
  # Search for one match but multiple found
  location_match = re.search(r'Location:\s(.+)', employees_str)
  print location_match.group(1) # US-MTV-CL5
  print ''
```

What if no matches are found?

If no matches are found, search() returns a NoneType object which does not have a group() method. So running the following will throw an error (NoneType does not have group method):

```
# Testing if a match is found
cc_match = re.search(r'Cost Center:(\d+)', employees_str)
cc_match.group(0)
```

The solution then, is to first test if a match was found with an if statement.

```
# Testing if a match is found
cc_match = re.search(r'Cost Center:(\d+)', employees_str)
if cc_match is not None:
  print cc_match.group(1)
else:
  print 'No Cost Center Found.'
```

Use `.search()` in place of `.findall()` when there is only 1 match to be found. Remember that the return types of the 2 methods are different. `MatchObject`s for `.search()` and lists of strings or tuples for `.findall()`.

Regex - Final thoughts

That is pretty much all we're going to say about regular expressions! The topic is actually MUCH more complex than this and extremely sophisticated expressions can be written, but most of the time, a simple pattern is all that is needed. I will be sending out a supplemental document with more Regex Goodies for those interested but none of that material will be covered in the final.

WEB CRAWLING

One of the most exciting features of python is its ability to open and read web pages. Create a new file called `google_crawl.py` and add the following:

```
import urllib #1

def UrlToText(url): #2
  url_file = urllib.urlopen(url) #3
  contents = url_file.read() #4
  return contents

def main():
  print UrlToText('https://www.google.com') #5

if __name__ == '__main__':
  main()
```

Run the script and look at the output. Now go to https://www.google.com and right click the page and select "View Source". Notice anything?

A few things to note here:

1. In order to work with webpages, you need to first import the `urllib` module
2. Instead of hardcoding this into the `main()` method, I am going to write a function instead that crawls a page and returns its contents. I want to do this since i'll be crawling multiple pages and doing the same thing.
3. The `urllib` module has a method called `urlopen` that actually takes multiple parameters, but we will only pass it 1 for now - the URL of the resource we want to access.
4. The `urlopen` method returns a network object that has a `.read()` method. The `.read()` method returns the contents of the webpage as a string.
5. In the main method, I'm going to call `UrlToText` on google.com

Combining Web Crawling with Regex

Regex rules supreme in environments where information is unstructured and formatting is unpredictable. Sounds like regex would work fantastic on webpages!

Crawling the google.com while easy, is actually not that useful. Instead let's crawl a resource that the GoogleWho team has been so gracious to provide us, the Orginfo API.

Orginfo API

First, try navigating to this page:

https://orginfo.corp.google.com/alberthwang?format=xml

In Firefox you should see the XML structure immediately, but in Chrome you need to right click and "View Source". In either browser, the content should look something like this (actually FF does a better job of formatting the XML page so I like to look at there):

```
<googler>
  <firstname>Albert</firstname>
  <lastname>Hwang</lastname>
  <title>Internal Tools Developer</title>
  <email>alberthwang</email>
  <manager>ninaye</manager>
  <cost_center>540: People Ops Admin</cost_center>
 .....
```

The OrgInfo API is a web-based protocol that exposes GoogleWho information in webpages. In this case:

```
https://orginfo.corp.google.com/{{ ldap }}?format=xml
```

Here, just by changing the {{ ldap }} field, we can pull GoogleWho information on any active Googler. Try it with your own page!

Regex on Orginfo API

What do you notice about the structure of the content of this page? All content is contained in XML tags like such:

```
<firstname>Albert</firstname>
```

This is prime real estate for regex! Now create a file called `orginfo_basic.py`:

```
import re
import urllib

def UrlToText(url):
  url_file = urllib.urlopen(url)
  contents = url_file.read()
  return contents

def main():
  my_moma_page = 'https://orginfo.corp.google.com/alberthwang?format=xml'
  url_contents = UrlToText(my_moma_page)
  print url_contents

if __name__ == '__main__':
  main()
```

Notice how python is able to grab the contents of your GoogleWho page. Fun, right?

Now change the content of your main() function to look like this:

```
  my_moma_page = 'https://orginfo.corp.google.com/alberthwang?
format=xml'
  employee_str =  UrlToText(my_moma_page)


cc_match = re.search('<cost_center_number>(\d+)</cost_center_number>',
                     employee_str)

  print cc_match.group(1)
```

This code will extract my **cost center number**.

Regex on Orginfo Searches

Next, try navigating to this page:

https://orginfo.corp.google.com/search/cost_center_number:540?format=xml

Moma-API support searches in this form:

```
https://orginfo.corp.google.com/search/{{ search params }}/xml
```

{{ search params }} is a set of concatenated keyword searches. In this case **cost_center_number:** corresponds to the cost center number. You can search most of the XML attributes shown in an individual page (firstname, displayname, employee_number, etc).

You can find a full list of different search techniques here:

http://goto/orginfo

Change the contents of your main function to this:

```
def main():
  cc540_page = 'https://orginfo.corp.google.com/search/'
  cc540_page += 'cost_center_number:540?format=xml'
  cc540_str =  UrlToText(cc540_page)

  ldaps = re.findall('<email>(\w+?)</email>', cc540_str)

  for ldap in ldaps:
    print ldap + ' is in Cost Center 540'
```

This will print all the ldaps of Googlers in the Cost Center 540.

Orginfo is just one example of fetching data resources through `urllib` and regex. One major advantage that crawling Orginfo has over pulling reports other reports is its flexibility and freshness. The data you pull is as recent as moma is updated. Also, leveraging the power of Python you can pull whatever you want and generate whatever kind of report you want.

CASE STUDY- Profile Crawling

This is an assignment I actually received from a client. He wanted to pull the contact information for all the Engineers listed on this website which is like a social site for open-source engineers.

http://www.advogato.org/person/

For each profile, he wanted the username, name, homepage, and profile link. There are 13,000 some odd profiles and it would take weeks to manually extract this data to source candidates.

I wrote a python script that scraped all 13,000 profiles instead of having people manually do the work. Can you guess how I did it?

“View Source” for the page and take a look.

What kind of regex patterns surround the username? The name?
What pattern dictates how the pages are organized when you click “Next page”?
Within each profile, what regex patterns would help in finding the homepage?

One feature request the client asked for was to get the level of each engineer too (Master, Apprentice, etc.). How would I go about extracting that?