CH 14: Other useful concepts

**14.1 Regular Expression:**

Regular expressions are a powerful language for matching text patterns. In this section, we will discuss basic introduction to regular expressions and how it can used in pattern matching. The Python "re" module provides regular expression support.

**14.1.1 Define pattern Language**

A regular expression or Pattern Language can be defines as special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

A **Fundamental Patten language** consists of following syntax.

|  |  |
| --- | --- |
| a, X, 9 | Ordinary characters just match themselves exactly. There are some special character, called meta characters, which do not match themselves because they have special meanings are: . ^ $ \* + ? { [ ] \ | ( ) |
| . (a period) | matches any single character except newline '\n' |
| \w (lowercase w) | matches a "word" character: a letter or digit or underbar [a - z A – Z, 0-9 and \_ ] |
| \W (upper case W) | matches any non-word character. |
| \b | boundary between word and non-Word |
| \s (lowercase s) | matches a single whitespace character like space, newline, return, tab, form [ \n\r\t\f]. |
| \S (upper case S) | matches any non-whitespace character. |
|  |  |
| \t, \n, \r | tab, newline, return |
| \d | decimal digit [0 to 9] |
| ^ | Matching at begging , |
| $ | Matching at end |
| \ | inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. |

**14.1.2. Introduction to re module**

Python provides very powerful RE module, which offers lot of pattern matching operation, which includes:

* **re.compile(pattern, flags=0) :** Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods.
* **re.search(pattern, string, flags=0)** - Scan through string, left to right, looking for a location where the regular expression pattern produces a match, and return a corresponding MatchObject instance. Return None if no position in the string matches the pattern
* **re.match(pattern, string, flags=0)** - If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance. Return None if the string does not match the pattern; note that this is different from a zero-length match.
* **re.split(*pattern*, *string*, *maxsplit=0*, *flags=0*)** - Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list.
* **re.findall(pattern, string, flags=0)** -Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups;
* **re.sub(*pattern*, *repl*, *string*, *count=0*, *flags=0*)** - Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn’t found, *string* is returned unchanged.

**14.1.3 Applying Regular expression with basic example:**

Let’s start with a simple example: We like to find out something in python having three h, that is “pythhhon”. We will apply re.search() method to do that. It takes two parameter, pattern and a string, in which to search, and return a pattern object if something found . Otherwise, it will return None. If it found something, then output of search() would be a match object and we can call match.group() to get the pattern.

import re

**Example1: Search for hhh in pythhhon** can be written as:

match = re.search(r'hhh', 'pythhhon') #=> found, match.group() == "hhh"

print match

print match.group()

**Example 2: Search for hhi in pythhhon** can be written as:

match = re.search(r'hhi', 'pythhhon') #=> not found, match == None

print match

**Example 3: Search for any 4 chracter ending with “ho” in pythhhon can be written as:**

## . = any char but \n

match = re.search(r'..ho', 'pythhhon') #=> found, match.group() == "hhho"

print match

print match.group()

**Example 4: Search for any 3 integers in 'pyt12345on'can be written as:**

## \d = digit char, \w = word char

match = re.search(r'\d\d\d', 'pyt12345on')# => found, match.group() == "123"

print match

print match.group()

**Example 5: Search for any 3 characters words in '**@@python!!’**can be written as:**

match = re.search(r'\w\w\w', '@@python!!')# => found, match.group() == "pyt"

print match

print match.group()

|  |  |
| --- | --- |
| Example | Description |
| . | Match any character except newline |
| \d | Match a digit: [0-9] |
| \D | Match a nondigit: [^0-9] |
| \s | Match a whitespace character: [ \t\r\n\f] |
| \S | Match nonwhitespace: [^ \t\r\n\f] |
| \w | Match a single word character: [A-Za-z0-9\_] |
| \W | Match a nonword character: [^A-Za-z0-9\_] |

In the above 5 example, shows, how we can do the search a pattern in a text. Some more complex example is given below:

There are 3 operation to test repetition , These are :

1. + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
2. \* -- 0 or more occurrences of the pattern to its left
3. ? -- match 0 or 1 occurrences of the pattern to its left

These operator are **Left & Greedy**, they search from left and return longest sequence witch match this pattern

**Example 6: Start with t and one or many h**

match = re.search(r'th+', 'pythhhon and pythhhon')# => found, match.group() == "thhh"

**Example 7: Start with t and zero or many h**

match = re.search(r'th\*', 'pyt and pythhhon')# => found, match.group() == "t"

**Example 8: Start with ‘t’ and zero or one ‘h’ and ‘on’ at end**

match = re.search(r'th?on', 'pyton and pythhhon')# => found, match.group() == "thhh"

**Example 11: Test begging with or ending with.**

We can use ^ or $ to test the begging or ending with. For example: **^py** match with anything start with **py** and **on$** match with anything with ends with **on**.

match = re.search(r'^py.\*', 'python')# => found, match.group() == "python"

match = re.search(r'^py.\*', 'tython')# => found, match.group() == None

match = re.search(r'.\*on$', 'python')# => found, match.group() == "python"

match = re.search(r'.\*on$', 'pythoT')# => found, match.group() == None

Some of the repetitive pattern example is as below:

|  |  |
| --- | --- |
| Example | Description |
| ruby? | Match "rub" or "ruby": the y is optional |
| ruby\* | Match "rub" plus 0 or more ys |
| ruby+ | Match "rub" plus 1 or more ys |
| \d{3} | Match exactly 3 digits |
| \d{3,} | Match 3 or more digits |
| \d{3,5} | Match 3, 4, or 5 digits |

We can do some kind of grouping in pattern. Suppose we want make a pattern , which can match “Python”, Jython and Hython. The pattern would be [PHJ]thon. Thus anyone in the pricket will be match.

**Example 9: Match Python, Jython or Hython**

match = re.search(r'[PHJ]ython', 'python')# => found, match.group() == "Python"

match = re.search(r'[PHJ]ython', 'Hython')# => found, match.group() == "Hython"

match = re.search(r'[PHJ]ython', 'Jython')# => found, match.group() == "Jython"

match = re.search(r'[PHJ]ython', 'Dython')# => not found,

**Example 10. Match Python, Jython or Hython, PPPython, JJJython,HHHython**

match = re.search(r'[PHJ]+ython', 'PPPython')# => found, match.group() == "Python"

here, [PHJ]+ indicate any number of P,H or J, thus PHHHHJJJP will be match with [PHJ]+

Similarly following class would works:

|  |  |
| --- | --- |
| Example | Description |
| [Pp]ython | Match "Python" or "python" |
| rub[ye] | Match "ruby" or "rube" |
| [aeiou] | Match any one lowercase vowel |
| [0-9] | Match any digit; same as [0123456789] |
| [a-z] | Match any lowercase ASCII letter |
| [A-Z] | Match any uppercase ASCII letter |
| [a-zA-Z0-9] | Match any of the above |
| [^aeiou] | Match anything other than a lowercase vowel |
| [^0-9] | Match anything other than a digit |

At this point let see what would be the regular expression if want to match with python or perl. These are called alternative word matching. Below is some of the example.

|  |  |
| --- | --- |
| Example | Description |
| python|perl | Match "python" or "perl" |
| rub(y|le)) | Match "ruby" or "ruble" |
| Python(!+|\?) | "Python" followed by one or more ! or one ? |

**Example 10. Match Python or Perl or Php**

>>> re.search('i love (python|perl|php)','i love python').group() #'i love python'

>>> re.search('i love (python|perl|php)','i love perl').group() #'i love perl'

>>> re.search('i love (python|perl|php)','i love php').group() #'i love php'

>>> re.search('i love (python|perl|php)','i love djnago').group() # Error

**14.1.4 Group Extraction using regular Expression**

Let’s write a regular expression which recognizes an email id:

>>> mystr= 'dipankar dutta.dipankar@gmail.com'

>>> match = re.search(r'[\w.-]+@[\w.-]+',mystr)

>>> print match.group()

[dutta.dipankar@gmail.com](mailto:dutta.dipankar@gmail.com)

In the above example, r'[\w.-][+@[\w.-](mailto:+@[\w.-)]+' indicate <any word including , and - >@<anyword including . and -> is a valid RE for email address. It will return the email-address present in mystr. Now suppose we want to extract user-id and mail-domain separrly. To solve this issue we need to grouping the extracted data. Python provide a way to do that.

We use first brackets to do that. If you want to extract Uid and mail domain seperatly , you should put those RE into breckets like this: **r'([\w.-]+)@([\w.-]+)'**

>>> match = re.search(r'([\w.-]+)@([\w.-]+)',mystr)

>>> match.group() #Whole Matches => 'dutta.dipankar@gmail.com'

>>> match.group(0) #Whole Match => 'dutta.dipankar@gmail.com'

>>> match.group(1) #UID: 'dutta.dipankar'

>>> match.group(2) #Mail Domain: 'gmail.com'

Thus using grouping you can extract individual matching data.

**14.1.4 Find all matching data.**

Let’s run the previous example for the following string.

>>> myStr = "dipankar dutta.ipankar@gmail.com, python python@python.com , Django Django@django.com"

>>> print re.search(r'[\w.-]+@[\w.-]+',mystr).group()

dutta.dipankar@gmail.com

>>>

As you can see that, It always return first match. What happen if I want to get all matches.

Python offers a module, called findall() similar to search, but it will return list of all matching data. For example:

>>> print re.findall(r'[\w.-]+@[\w.-]+',mystr)

['dutta.dipankar@gmail.com', 'hello@hello.com', 'amazon@amazon.com']

>>>

We can also apply grouping to get all UID and Mail-domain.

>>> print re.findall(r'([\w.-]+)@([\w.-]+)',mystr)

[('dutta.dipankar', 'gmail.com'), ('hello', 'hello.com'), ('amazon', 'amazon.com')]

As you can see we get only UID and mail-doamin by Email Id got lossed, to retrieve email id we can put another brackets to cover whole regular expression.

>>> print re.findall(r'(([\w.-]+)@([\w.-]+))',mystr)

[('dutta.dipankar@gmail.com', 'dutta.dipankar', 'gmail.com'), ('hello@hello.com', 'hello', 'hello.com'), ('amazon@amazon.com', 'amazon', 'amazon.com')]

**14.1.4 Non-greedy matching.**

As you see regular expression is very helpful to perse the HTML page to gather data. Suppose we have following html code

>>> html= """

<html>

<body>

<p> hello world1 </p> <p>helloWorld2 </p> <p>helloWorld3 </p>

</body>

</html>

"""

>>> html

'\n<html>\n<body>\n<p> hello world1 </p> <p>helloWorld2 </p> <p>helloWorld3 </p>\n</body>\n</html>\n'

Now, we want to find out all text inside <p> blocks, the code might be looks like:

>>> print re.findall(r'<p>.\*</p>',html)

['<p> hello world1 </p> <p>helloWorld2 </p> <p>helloWorld3 </p>']

But, As you see that it will return longest match, do you know why ? This is because \*, + works LEFT and GREEDY, hence always want to consume longest match. To make it non gready we can put a ‘?’ after \*. The correct RE is as below:

>>> print re.findall(r'<p>.\*?</p>',html)

['<p> hello world1 </p>', '<p>helloWorld2 </p>', '<p>helloWorld3 </p>']

>>>

It is Still containing <p> tags, we can remove this using grouping, Hence final code would be :

>>> print re.findall(r'<p>(.\*?)</p>',html)

[' hello world1 ', 'helloWorld2 ', 'helloWorld3 ']

Below are the references, which can work for non-greedy:

|  |  |
| --- | --- |
| <.\*?> | Nongreedy: matches "<python>" in "<python>perl>" |
| <.+?> | Nongreedy: matches "<python>" in "<python>perl>" |

**14.1.4 Other RE facilities**

**A. Replacement:** Re module allow a way to replace all matched text by some other value. We can use re.sub to do that, Syntax is :

re.sub(pat, replacement, str) -- returns new string with all replacements,

Going back to our Email example, we want replace all mail domain by django.com, the code would be like:

>>> myStr

'dipankar dutta.ipankar@gmail.com, python python@python.com , Django Django@django.com'

>>> print re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@django.com', myStr)

dipankar dutta.ipankar@django.com, python python@django.com , Django [Django@django.com](mailto:Django@django.com)

**A. Match Module:** it is similar as of search module, but it will do the search from beginning. For example:

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line) # Matches

matchObj = re.match( r'dogs', line) # Does not matched

matchObj = re.search( r'dogs', line) # Matches

**14.2 Logging in Python.**

In reality, logging is very important. When you transfer money, from one account to another, you must logging in some note-books. When an airplane is flying, black box is recording everything. If something goes wrong, people can read the note-book or replay the record and has a chance to figure out what happened. Likewise, logging is important for software developing, debugging and running. When a program crashes, if there is no logging record, you have little chance to understand what happened. Without the log, you can hardly know what’s wrong if a service goes down or some service call fails. Not only for the web -servers, logging is equally important for desktop GUI applications. For example, when your program crashes on your customer’s PC, you can ask them to send the log files to you, and you may be able to figure why, otherwise it might be very difficult to reproduce to same issue in your local system.

Every programming language provides own logging facilities. Python provide a logging module to help to log software event.

14.2.1 **Print is not a good idea**

Although logging is important, not all developers know how to use them correctly. Some developers insert print statements when developing and remove those statements when it is finished. It may looks like this

print 'Start reading database'

records = model.read\_recrods()

print '# records', records

print 'Updating record ...'

model.update\_records(records)

print 'done'

Two main drawback of this:

1. There is no way to know what’s happen in production as print are removed.
2. While removing these print, there is a chance to remove some important data
3. If you miss some print statement to remove, it will display unwanted messages.

**14.2.2 Introduction to logging module.**

Python provide a better way to do this logging. It’s a standard module, well designed to be easy to use and very flexible. The first important thing to realize about the logging package is that there are four types of objects that each plays an important role in getting the logging system working:

* Loggers – used by application code to send messages to the logging system.
* Formatters – format the message for output.
* Filters – provide fine-grained output control.
* Handlers – send formatted output to a destination such as a file.

A very basic example is as below:

import logging #Import logging

logging.basicConfig(level=logging.INFO) #Set config

logger = logging.getLogger(\_\_name\_\_) # get logger name

logger.info('Start reading database')

# read database here

records = {'python': 2, 'django': 3}

logger.debug('Records: %s', records)

logger.info('Updating records.')

# update records here

logger.info('Finish updating records')

Output:

INFO:\_\_main\_\_:Start reading database

INFO:\_\_main\_\_:Updating records.

INFO:\_\_main\_\_:Finish updating records

As you see the debug messge is not printed yet. Logging has 3 basic advantage on”print approach”

1. Control message level and filter out unwanted message.
2. Display the important infowmation while displaying message, like whicg module produce this or this event occure
3. Where to display this message or store it some where.

Subsequent section **we** will discuss on this.

**14.2.3. Control message level.**

Python logging module provides different importance levels you can use: debug, info, warning, error and critical. By giving different level to logger, you can write only some messages to specific log file. The level are in following order: debug < info < warning < error < critical. That means, it I set a log level to info, all message will get printed except debug. If I set log level as critical, obly critical message get printed. If I spacify lo level as DEBUG, it will print all message.

Below Example show this:

Set the log level of INFO, it would print everything other than debug

import logging #Import logging

logging.basicConfig(level=logging.INFO) #Set config

logger = logging.getLogger(\_\_name\_\_) # get logger name

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

**Output:**

INFO:\_\_main\_\_:info message

WARNING:\_\_main\_\_:Checkout this warning.

ERROR:\_\_main\_\_:An error goes here.

CRITICAL:\_\_main\_\_:Something critical happened.

Set the log level as DEBUG, It will print every message

import logging #Import logging

logging.basicConfig(level=logging.DEBUG) #Set config

logger = logging.getLogger(\_\_name\_\_) # get logger name

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

Output:

DEBUG:\_\_main\_\_:debug message

INFO:\_\_main\_\_:info message

WARNING:\_\_main\_\_:Checkout this warning.

ERROR:\_\_main\_\_:An error goes here.

CRITICAL:\_\_main\_\_:Something critical happened.

Set the log level as ERROR, it will only print ERROR and CRITICAL

import logging #Import logging

logging.basicConfig(level=logging.ERROR) #Set config

logger = logging.getLogger(\_\_name\_\_) # get logger name

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

Output:

ERROR:\_\_main\_\_:An error goes here.

CRITICAL:\_\_main\_\_:Something critical happened.

**14.2.4. Formatting Log messages:**

Python logger also allow us to formet log message: Like if we want to see which module produce this log and when it go produced, we can formet log message in that way. The below example shows how we can add a logging formetter

import logging #Import logging

logger = logging.getLogger('myfunc') # get logger name

# Set logging level

logger.setLevel(logging.DEBUG)

# create a FORMAT and set the FORMAT for the handler.

FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

logging.basicConfig(format=FORMAT)

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

OUTPUT:

2013-12-07 16:00:31,342 - myfunc - DEBUG - debug message

2013-12-07 16:00:31,362 - myfunc - INFO - info message

2013-12-07 16:00:31,371 - myfunc - WARNING - Checkout this warning.

2013-12-07 16:00:31,378 - myfunc - ERROR - An error goes here.

2013-12-07 16:00:31,384 - myfunc - CRITICAL - Something critical happened.

>>>

You can also change the date format as below:

import logging #Import logging

logger = logging.getLogger('myfunc') # get logger name

# Set logging level

logger.setLevel(logging.DEBUG)

# create a FORMAT and set the FORMAT for the handler.

FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

logging.basicConfig(format=FORMAT,level=logging.INFO,datefmt='%Y-%m-%d %H:%M:%S')

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

>>> Output <<<

2013-12-07 16:03:22 - myfunc - DEBUG - debug message

2013-12-07 16:03:22 - myfunc - INFO - info message

2013-12-07 16:03:22 - myfunc - WARNING - Checkout this warning.

2013-12-07 16:03:22 - myfunc - ERROR - An error goes here.

2013-12-07 16:03:22 - myfunc - CRITICAL - Something critical happened.

**14.2.4. Where to put log messages: LogHandaler**

The Handler is responsible for outputting your message - to a file, stream, socket or where ever you want it to go. You can add a Handler to the Logger by using the addHandler method of the Logger. A Logger can have multiple Handlers attached. There are a lot of custom handlers (like FileHandler, RotatingFileHandler, HTTPHandler, etc.) in the logging module. Use them to make your life easier.

In previous example we found that the log message got printed in console, Let’s also store the log messages in some file :

import logging #Import logging

logger = logging.getLogger('myfunc') # get logger name

# Set logging level

logger.setLevel(logging.DEBUG)

# create a FORMAT and set the FORMAT for the handler.

FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

logging.basicConfig(format=FORMAT,level=logging.INFO,datefmt='%Y-%m-%d %H:%M:%S')

# add a file handler

fh = logging.FileHandler('myapp.log')

fh.setLevel(logging.WARNING)

# create a formatter and set the formatter for the handler.

frmt = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

fh.setFormatter(frmt)

# add the Handler to the logger

logger.addHandler(fh)

# You can now start issuing logging statements in your code

logger.debug('debug message') # This won't print

logger.info('info message') # Neither will this.

logger.warn('Checkout this warning.') # This will show up in the log file.

logger.error('An error goes here.') # and so will this.

logger.critical('Something critical happened.') # and this one too.

OutPut:

As you can see the logs got printed in the console as well as written in the log file.

Open myapps.log and you will find out the following log in it

2013-12-07 16:10:44,763 - myfunc - WARNING - Checkout this warning.

2013-12-07 16:10:44,770 - myfunc - ERROR - An error goes here.

2013-12-07 16:10:44,776 - myfunc - CRITICAL - Something critical happened.

**14.2.5. Log-rotate**

import glob

import glob

import logging

import logging.handlers

import time

LOG\_FILENAME = 'python.log'

LOGGING\_MSG\_FORMAT = '%(name)-14s > [%(levelname)s] [%(asctime)s] : %(message)s'

LOGGING\_DATE\_FORMAT = '%Y-%m-%d %H:%M:%S'

logging.basicConfig(

level=logging.DEBUG,

format=LOGGING\_MSG\_FORMAT,

datefmt=LOGGING\_DATE\_FORMAT

)

root\_logger = logging.getLogger('')

handler = logging.handlers.TimedRotatingFileHandler(LOG\_FILENAME,'S',1)

handler.suffix = "%Y-%m-%d" # or anything else that strftime will allow

root\_logger.addHandler(handler)

while True:

daemon\_logger = logging.getLogger('TEST')

daemon\_logger.info("Puting one Logs...")

time.sleep(6)

# See what files are created

logfiles = glob.glob('%s\*' % LOG\_FILENAME)

for filename in logfiles:

print(filename)

**14.2.5 Logging Best Practices:**

1. Use \_\_name\_\_ as the logger name: while creating logger object, you can pass any string for the name, but it is always advisable to specify \_\_name\_\_. It is automatically return the full path name, including package, module class or function name.
2. Capture exceptions and record them with trace-back: It is always a good practice to record when something goes wrong, but it won’t be helpful if there is no traceback. So always put a log message in every except part in a try block
3. Use JSON or YAML logging configuration: You can configure your logging system in Python code, but it is not flexible. It’s better to use a logging configuration file. After Python 2.7, you can load logging configuration from a dict. It means you can load the logging configuration from a JSON or YAML file. See more about this on python Docs
4. Use rotating file handler: If you use FileHandler for writing logs, the size of log file will grow with time. Someday, it will occupy your entire disk. In order to avoid that situation, you should use RotatingFileHandler instead of FileHandler in production environment.
5. Setup a central log server when you have multiple servers: When you have multiple servers and different log files. You can setup a central log system to collect all important (warning and error messages in most cases). Then you can monitor it easily and notice what’s wrong in your system.
6. Write logging records everywhere with proper level: With flexibility of the logging module, you can write logging record everywhere with proper level and configure them later.

**14.3. Accessing web using urllib2**

Somwtime you might require to download a webpage or login to a site using programing and doing some automated task. For instance, you might need to do some add blog post using script, where the text should taken from a file. Or download a full site for offline read or want to send some sms through some free sms sending site everyday morning, automatically.

Python has a wonderful library, called ulrlib2, for doing these HTTP task. While we talking about urllib2, we are going to show some examples where you use urllib, since they are often used together.

Going forword, we will going to discuss on Gtting URLs, Requests, Posts, User Agents and Error handling.

Some definition of basic technology is defined as below:

* HTTP is based on requests and responses - the client makes requests and servers send responses back to client
* A program on the Internet can work as a client (access resources) or like a server (makes services available).
* An URL identifies a resource on the Internet, uniquely.

**14.3.1 Fetching an URL and Print data and header**

urllib2 offers a very simple interface, in the form of the urlopen function. This function is capable of fetching URLs using a variety of different protocols like HTTP, FTP etc. it will return a response object. We can use resp.read() to get the HTML and resp.info() return the header info.

Syntax : urllib2.urlopen(url[, data][, timeout])

Below Example Show the details:

import urllib2

resp = urllib2.urlopen('http://python.org/')

html = resp.read()

print html

print resp.info()

resp.close()

Output of resp.info() as below

Date: Sat, 07 Dec 2013 13:35:05 GMTServer: Apache/2.2.16 (Debian)Last-Modified: Fri, 06 Dec 2013 18:03:52 GMTETag: "105800d-50eb-4ece178017200"Accept-Ranges: bytesContent-Length: 20715Vary: Accept-EncodingConnection: closeContent-Type: text/html

Following program , fetch the url and save in a file:

import urllib2

file = "downloaded.html" # file to be written to

url = "<http://www.python.com/>"

response = urllib2.urlopen(url)

#open the file for writing, read from request while writing to file

f = open(file, "w")

f.write(response.read())

f.close()

To save a binary file use “wb” option when opening afile :

import urllib2

mp3file = urllib2.urlopen("<http://www.example.com/songs/mp3.mp3>")

output = open('test.mp3','wb') # open a fine in write binary mode

output.write(mp3file.read())

output.close()

Flowing program will print different information about this request and response.

import urllib2

response = urllib2.urlopen('http://python.org/')

print "Response:", response

###############################

print "The URL is: ", response.geturl() # Get the URL. This gets the real URL.

print "This gets the code: ", response.code # Getting the code

print "The Headers are: ", response.info() # Get the Headers,This returns a dictionary-like object

print "The Date is: ", response.info()['date'] # Get the date part of the header

print "The Server is: ", response.info()['server'] # Get the server part of the header

html = response.read() # Get all data , Will get HTML data

print "Get all data: ", html

**14.3.1 A better way to access Web: GET vs POST**

Unlike doing urlopen(url), We can prepare the request and then send the request to the serevr for response. When I am saying “request”, it is not only the url, but also contains a number of thing: header info, HTTP GET request parameter. POST request pavement or some file object we want to send, and cookie information.

Let prepare a request, which contains all these associative data and then send to server. This can be done by urllib2.Request It will return a request object which is pass into urlopen function.

Syntax : class urllib2.Request(url[, data][, headers][, origin\_req\_host][, unverifiable])

url - should be a string containing a valid URL.

data - may be a string specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use data; the HTTP request will be a POST instead of a GET when the data parameter is provided. data should be a buffer in the standard application/x-www-form-urlencoded format. The urllib.urlencode() function takes a mapping or sequence of 2-tuples and returns a string in this format.

Headers - should be a dictionary, and will be treated as if add\_header() was called with each key and value as arguments. This is often used to “spoof” the User-Agent header.

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

Send a request with modified header: This says the request is coming from Mozilla

import urllib2

url = '<http://www.google.com/>#q=my\_search'

headers = {'User-Agent' : 'Mozilla 5.10'} ## Add your headers just like dict’

request = urllib2.Request(url, None, headers) ## Create the Request.

request.add\_header('User-agent', 'Mozilla 5.10') # You can this kind of operaion as well.

response = urllib2.urlopen(request) # Getting the response

print response.headers # Print the headers

Send a GET request with some parameters. When we don't include the data (and only pass the url), the request being made is actually a GET request

import urllib

import urllib2

url = 'http://www.google.com/'

values = {'q' : 'hello python','lang':'english','opt':'!@#$%^&\*()' }

data = urllib.urlencode(values)

print data # q=hello+python&lang=english&opt=%21%40%23%24%25%5E%26%2A%28%29

url = url + ‘?’+data

req = urllib2.Request(url)

response = urllib2.urlopen(req)

print request.geturl() #http://www.google.com/?q=hello+python&lang=english&opt=%21%40%23%24%25%5E%26%2A%28%29

let’s send a POST request with some parameters. When you do include the data in urllib2.Request() , the request being made is a POST request, where the url will be your post url, and the parameter will be http post content.

import urllib

import urllib2

url = 'http://www.google.com/'

values = {'q' : 'hello python','lang':'english','opt':'!@#$%^&\*()' }

data = urllib.urlencode(values)

print data # q=hello+python&lang=english&opt=%21%40%23%24%25%5E%26%2A%28%29

req = urllib2.Request(url,data)

response = urllib2.urlopen(req)

print request.geturl() #http://www.google.com as it is a POST request

**14.3.1 urllib.urlparse**

The urlparse module provides functions to analyze URL strings. It defines a standard interface to break Uniform Resource Locator (URL) strings up in several optional parts, called components, known as (scheme, location, path, query and fragment)

URL = scheme + netloc<url:port>+Path

>>> url = 'http://www.google.com:8080/pages/index.html'

>>> import urlparse

>>> print urlparse.urlsplit(url)

SplitResult(scheme='http', netloc='www.google.com:8080', path='/pages/index.html', query='', fragment='')

**14.3.4 urlib.urlencode()**

When you pass information through a URL, you need to make sure it only uses specific allowed characters. Allowed characters are any alphabetic characters, numerals, and a few special characters that have no meaning in the URL string but some character has special meaning in url ( to represent the url ) for example: space or a + character. Arguments can be passed to the server by encoding them with and appending them to the URL. Urllib.urlencode does this facility, show in given example:

values = {'q' : 'hello python','lang':'english','opt':'!@#$%^&\*()' }

data = urllib.urlencode(values)

print data # q=hello+python&lang=english&opt=%21%40%23%24%25%5E%26%2A%28%29

**14.3.5 Cookie handling**

Cookie is just another HTTP header. Here I have used urllib2.build\_opener() to create an request and the add the cookie in it, at last I have used opener.open() to request to the serevr.

import urllib2

opener = urllib2.build\_opener()

opener.addheaders.append(('Cookie', 'cookiename=cookievalue'))

f = opener.open("http://google.com/")

Note that, the cookies you get previously are automatically sent again in following requests.

There are more ways how to handle cookies. Some modules like [cookielib](http://docs.python.org/library/cookielib.html) try to behave like web browser.

**14.3.5 Example: Grabbing a full Site:**

**Example1:** A crawler is a program that starts with a url on the web (ex: http://python.org), fetches the web-page corresponding to that url, and parses all the links on that page into a repository of links. Next, it fetches the contents of any of the url from the repository just created, parses the links from this new content into the repository and continues this process for all links in the repository until stopped or after a given number of links are fetched.

**Solution :**  the maim of this exercise is to retrieve n number of unique urls for python.org.

import urllib2

import re

def fetchData(url):

""" Input - url

output - list of tuple <internal url and hyperlinktext>

"""

try:

print 'getting',url

resp = urllib2.urlopen(url)

html = resp.read()

#Regular expression to get all link

linkList = re.findall(r'<a .\*?href="(.\*?)".\*?>(.\*?)<',html)

#Make all link are absulate

abslink = [ ]

for l,t in linkList: # Not a base URL

if not t.startswith('http') and t.endswith('/'):

abslink.append((url+l,t))

elif 'python.org' in l : # Not in other Domain hence filter it out

abslink.append((l,t))

return abslink

except:

print 'OOPS.. errro retiving file'

return []

def getUrlRepo(baseurl,count):

"""

input : BAse url and number of url to be returned

output : list of touple < url,text> of length count

"""

repo =[]

unique\_dict={}

unique\_dict[baseurl]='python'

repo.append((baseurl,'python'))

process\_index = 0

count = count -1;

while count > 0:

cur\_list = fetchData(repo[process\_index][0])

process\_index = process\_index + 1

for lnk,txt in cur\_list:

if lnk not in unique\_dict:

unique\_dict[lnk] = txt

repo.append((lnk,txt))

count = count -1

if count < 0:

break

return repo

###########################################################

x =raw\_input('Enter how many link u want: ')

data = getUrlRepo('http://python.org/',int(x))

for i,d in enumerate(data):

print i,'.',d[0],':',d[1]

**14.2.2 Introduction to threading module**

**14.5 Send Mail using python**

It’s easy to send a mail using python. Python provide a very useful module, called smtplib, to achive this. In this section, how you can access your gmail account using smtplib. Here we will explore, how you can retrieve a mail, send mail and send a mail with attachment.

**14.5.1. logging to your Gmail account.**

>>> import smtplib

>>> session = smtplib.SMTP('smtp.gmail.com', 587)

>>> session.ehlo()

(250, 'mx.google.com at your service, [101.212.1.176]\nSIZE 35882577\n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')

>>> session.starttls()

(220, '2.0.0 Ready to start TLS')

>>> session.ehlo()

(250, 'mx.google.com at your service, [101.212.1.176]\nSIZE 35882577\n8BITMIME\nAUTH LOGIN PLAIN XOAUTH XOAUTH2 PLAIN-CLIENTTOKEN\nENHANCEDSTATUSCODES\nCHUNKING')

>>> session.login('dutta.dipankar08@gmail.com', 'WrongPass')

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

session.login('dutta.dipankar08@gmail.com', 'XXXXX’)

File "C:\Python27\lib\smtplib.py", line 615, in login

raise **SMTPAuthenticationError**(code, resp)

SMTPAuthenticationError: (535, '5.7.8 Username and Password not accepted. Learn more at\n5.7.8 http://support.google.com/mail/bin/answer.py?answer=14257 sg1sm6013633pbb.16 - gsmtp')

>>> session.login('dutta.dipankar08@gmail.com', ‘CorrectPassword’)

(235, '2.7.0 Accepted')

#### You can do all operation Here ####

SMTPServerDisconnected: Connection unexpectedly closed: [Errno 10053] An established connection was aborted by the software in your host machine

**14.5.3 Sending a Mail**

import smtplib

SMTP\_SERVER = 'smtp.gmail.com'

SMTP\_PORT = 587

sender = 'sender@gmail.com'

recipient = 'recicver@example.com'

subject = 'Sample Gmail SMTP Test'

body = 'This is a sample body !!!'

headers = ["From: " + sender,

"Subject: " + subject,

"To: " + recipient,

"MIME-Version: 1.0",

"Content-Type: text/html"]

headers = "\r\n".join(headers)

session = smtplib.SMTP(server, port)

session.ehlo()

session.starttls()

session.ehlo

session.login(sender, password)

session.sendmail(sender, recipient, headers + "\r\n\r\n" + body)

session.quit()

**14.5.4 Sending a mail with attachments**

import smtplib

from email.MIMEMultipart import MIMEMultipart

from email.MIMEBase import MIMEBase

from email.MIMEText import MIMEText

from email import Encoders

import os

gmail\_user = "your\_email@gmail.com"

gmail\_pwd = "your\_password"

def mail(to, subject, text, attach):

msg = MIMEMultipart()

msg['From'] = gmail\_user

msg['To'] = to

msg['Subject'] = subject

msg.attach(MIMEText(text))

part = MIMEBase('application', 'octet-stream')

part.set\_payload(open(attach, 'rb').read())

Encoders.encode\_base64(part)

part.add\_header('Content-Disposition',

'attachment; filename="%s"' % os.path.basename(attach))

msg.attach(part)

session = smtplib.SMTP("smtp.gmail.com", 587)

session.ehlo()

session.starttls()

session.ehlo()

session.login(gmail\_user, gmail\_pwd)

session.sendmail(gmail\_user, to, msg.as\_string())

# Should be mailServer.quit(), but that crashes...

session.close()

mail("some.person@some.address.com",

"Hello from python!",

"This is a email sent with python",

"my\_picture.jpg")

**14.5.2. Reading your mail Box**

The IMAP protocol document is absolutely key to understanding the commands available, but let me skip attempting to explain and just lead by example where I can point out the common gotchas I ran into.

Let’s logging to Gmail account using IMAP

UNAME = 'dutta.dipankar08@gmail.com'

PASSWD = ‘XXXX’

import imaplib

mail = imaplib.IMAP4\_SSL('imap.gmail.com')

mail.login(UNAME, PASSWD)

mail.list()

# Out: list of "folders" aka labels in gmail.

mail.select("inbox") # connect to inbox.

Now let’s get the list of email in box and print the latest one:

result, data = mail.search(None, "ALL")

ids = data[0] # data is a list.

id\_list = ids.split() # ids is a space separated string

latest\_email\_id = id\_list[-1] # get the latest

result, data = mail.fetch(latest\_email\_id, "(RFC822)") # fetch the email body (RFC822) for the given ID

raw\_email = data[0][1] # here's the body, which is raw text of the whole email

# including headers and alternate payloads

The imap search function returns a sequential id, meaning id 5 is the 5th email in your inbox.

That means if a user deletes email 10, all emails above email 10 are now pointing to the wrong email. This is unacceptable. Luckily we can ask the imap server to return a UID (unique id) instead. Let’ get the latest email using UIDs instead of volatile sequential ids:

result, data = mail.uid('search', None, "ALL") # search and return uids instead

latest\_email\_uid = data[0].split()[-1]

result, data = mail.uid('fetch', latest\_email\_uid, '(RFC822)')

raw\_email = data[0][1]

You can see emails are printed in above program look like gibberish. Python offers a module, called email, which can be used for reading email properly.

import email

email\_message = email.message\_from\_string(raw\_email)

print email\_message['To']

print email.utils.parseaddr(email\_message['From'])

print email\_message.items() # print all headers

# print body..

maintype = email\_message.get\_content\_maintype()

if maintype == 'multipart':

for part in email\_message.get\_payload():

if part.get\_content\_maintype() == 'text':

return part.get\_payload()

elif maintype == 'text':

return email\_message.get\_payload()

**14.6 Multithreading in python**

When you run a normal Python script, the program execution starts at the first line and goes down line by line. Loops and function calls may cause the program execution to jump around, but it is fairly easy to see from the code which line will get executed next at any given point. You can put your finger on the first line of code in the .py file on the screen, and then trace through the next lines of code that are executed. This is **single-threaded programming**.

Whereas, using multiple threads, is like putting a second finger down on your code. Each finger still moves the same way, but now they are executing code simultaneously.

**14.16.1 Advantage of Multi-threading:**

You can start dozens or hundreds of threads in your program (that’s a lot of fingers). This doesn’t automatically make your programs dozens or hundreds of times faster though (all the threads are still sharing the same CPU) but it can make your program more efficient.

**14.6.2 Example of Multi-threading and why it is important !**

Suppose we want to develop n downloader application, where we want to download a file full of name of cities, then sort the cities, and then writes these cities to a file on our computer. If there are thousands of files needs to process, we would put a call to this function in a loop and it would handle each file serially: <download, sort, write>, <download, sort, write>, download, sort, write and so on. The code will be like this

url\_list=[]

for u in urlist:

data = download(url)

data = sorted(data)

write(data)

The above approach is absolutely correct but there is a serious drawback in term of performance. If you note, each of these three steps use different resources on your computer: downloading uses the network connection, sorting uses the CPU, writing the file uses the hard drive. And these things happens in different speed, when some file got downloaded, that that time CPU become idle and I/O operation is also not taking place. It would better if you have three workers on these parallel, such that efficiency got improved.

To improve this situation, we can call this function hundreds of times in parallel by using one thread for each file. This will not only make better use of the bandwidth, but if some files download sooner than others, the CPU can be used to sort them while the network connection continues to work. This makes more efficient use of your computer.

There are two ways to create the threads in python, discussed below:

**14.6.3 Multithreading using thread module**

Using thread module, we can start a thread by calling start\_new\_thread() function, which takes function name , which to be execute and the paramets , which neeed to pass to the function. Below example a creted a function print\_time(), which basically counting number from 1 to infinity after some interval of time. Then I have created three threaded, which count numbers <1 to infinity > simultaneously.

import thread

import time

# Define a function for the thread

def print\_time( threadName, sleeptime):

count = 0

while count < 5:

time.sleep(sleeptime)

count += 1

print "%s: %s : Counting %d" % ( threadName, time.ctime(time.time()),count )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread2", 4, ) )

thread.start\_new\_thread( print\_time, ("Thread3", 6, ) )

except:

print "Error: unable to start thread"

while 1:

pass

A sample output is as below, As you see counting by three threads are interleaved in between.

Thread1: Sun Dec 08 13:27:13 2013 : Counting 1

Thread2: Sun Dec 08 13:27:15 2013 : Counting 1

Thread3: Sun Dec 08 13:27:15 2013 : Counting 1

Thread1: Sun Dec 08 13:27:15 2013 : Counting 2

Thread1: Sun Dec 08 13:27:17 2013 : Counting 3

Thread2: Sun Dec 08 13:27:19 2013 : Counting 2

Thread3: Sun Dec 08 13:27:19 2013 : Counting 2

Thread1: Sun Dec 08 13:27:19 2013 : Counting 4

Thread1: Sun Dec 08 13:27:21 2013 : Counting 5

Thread2: Sun Dec 08 13:27:23 2013 : Counting 3

Thread3: Sun Dec 08 13:27:23 2013 : Counting 3

Thread2: Sun Dec 08 13:27:27 2013 : Counting 4

Thread3: Sun Dec 08 13:27:27 2013 : Counting 4

**14.6.3 Multithreading using threading module**

Still, using thread module, we can create number of threads efficiently, still they are limited by functionality, which is basically removed by new threading module, which is supported by python2.4 onward.

Using threading module we can create threads by following steps:

1. Define a your customize thred class , which is a subclass of the thread class
2. Override the \_\_init\_\_ method by adding your customize argumnets
3. Override the run() method to implement the function, what a thread suppose to do.

The basic method provided my Thread class is as below:

run() - The run() method is the entry point for a thread.

start()- The start() method starts a thread by calling the run method.

join([time])- The join() waits for threads to terminate.

isAlive()- The isAlive() method checks whether a thread is still executing.

getName()- The getName() method returns the name of a thread.

setName()- The setName() method sets the name of a thread.

The same example of the previous example using threading module as below:

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, threadName,sleeptime):

"Will take <threadId,name and sleeptime>"

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.threadName = threadName

self.sleeptime = sleeptime

def run(self):

print "Starting " + self.threadName +'\n'

count = 0

while count < 5:

time.sleep(self.sleeptime)

count += 1

print "%s: %s : Counting %d" % ( self.threadName, time.ctime(time.time()),count )

print "Exiting " + self.threadName +'\n'

# Create new threads

thread1 = myThread(1, "Thread-1",2)

thread2 = myThread(2, "Thread-2",4)

thread3 = myThread(3, "Thread-2",6)

# Start new Threads

thread1.start()

thread2.start()

thread3.start()

print "Exiting Main Thread"

The above example will show the same out as before.

Output :

Starting Thread-2

Exiting Main Thread

Starting Thread-2

Starting Thread-3

Thread-1: Sun Dec 08 13:53:02 2013 : Counting 1

Thread-3: Sun Dec 08 13:53:03 2013 : Counting 1

Thread-1: Sun Dec 08 13:53:04 2013 : Counting 2

Thread-2: Sun Dec 08 13:53:05 2013 : Counting 1

Thread-1: Sun Dec 08 13:53:06 2013 : Counting 3

**14.6.3 Wait for other thread to join : join()**

As you see, 4 threaded is stared in parallel, one main thread and other three are just created from main thread. Only problem is that main thread is exited before any other threads, as the thread are operated independently. In ideal situation, the main thread should wait until all thread are completed their task. We can use join() function to do that.

Modified code is as below:

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, threadName,sleeptime):

"Will take <threadId,name and sleeptime>"

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.threadName = threadName

self.sleeptime = sleeptime

def run(self):

print "Starting " + self.threadName +'\n'

count = 0

while count < 5:

time.sleep(self.sleeptime)

count += 1

print "%s: %s : Counting %d\n" % ( self.threadName, time.ctime(time.time()),count )

print "Exiting " + self.threadName +'\n'

# Create new threads

thread1 = myThread(1, "Thread-1",2)

thread2 = myThread(2, "Thread-2",4)

thread3 = myThread(3, "Thread-3",6)

#craeatng a thread list

threadlist=[]

threadlist.append(thread1);

threadlist.append(thread2);

threadlist.append(thread3);

# Start new Threads

thread1.start()

thread2.start()

thread3.start()

#main thread shoudl wait for all thread to join.

for t in threadlist:

t.join()

print "Exiting Main Thread"

Now the output is as below

>>>

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1: Sun Dec 08 14:08:19 2013 : Counting 1

Thread-2: Sun Dec 08 14:08:21 2013 : Counting 1

Thread-1: Sun Dec 08 14:08:21 2013 : Counting 2

Thread-3: Sun Dec 08 14:08:23 2013 : Counting 1

Thread-1: Sun Dec 08 14:08:23 2013 : Counting 3

Exiting Thread-1

Thread-2: Sun Dec 08 14:08:25 2013 : Counting 2

Thread-3: Sun Dec 08 14:08:29 2013 : Counting 2

Thread-2: Sun Dec 08 14:08:29 2013 : Counting 3

Exiting Thread-2

Thread-3: Sun Dec 08 14:08:35 2013 : Counting 3

Exiting Thread-3

Exiting Main Thread <<< See at end

>>>

**14.6.2 Theoretical concept about Synchronization:** Let discussed, some of the important concept of thread sync before going to details.

*1. What is atomic operation ?*In simple terms, atomicity is unbreakability, i.e. an uninterrupted operation. If two users issue a print command, each print should go in single attempt. If the printer driver is sending parts of data from two users, the printout will not be as expected. Hence, the printer driver must send the print command as unbreakable operation from one application at a time. This is kind of all or None concept.

*2. What is critical section?* In simple terms a critical section is group of instructions/statements or region of code that needs to be executed atomically such as accessing a resource (file, input or output port, global data, etc. In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. race across threads), the result is unpredictable. To avoid such situation, a thread must acquire a lock prior to executing critical section. The lock can be acquired by only one thread.

3. *What is producer- consumer problem?* In computing, the producer–consumer is a classic example of a multi-process synchronization problem. Suppose we have two processes, the producer and the consumer, who share a common, fixed-size list used as a queue. The producer generates a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the list) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

4. *What is synchronization service two process/ threads?* It’s a method to provide a way to synchronize between two or more process while accessing shared resource or a critical section. Like other languages, python support the way to synchronize.

*5. What is Mutual exclusion?* Ensuring to access shared resources one at a time.

*6. What is Mutex or Locks*? A mutex provides mutual exclusion; either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

*7. What is semaphore?:* A semaphore is a generalized mutex. In lieu of single list of size 100, we can split the 4 list having max 25 items. A semaphore can be associated with these four lists. The consumer and producer can work on different lists at the same time.

*8. What is Binary Semaphore?* Just acts like mutex when we have one list.

*9. Then, Binary Semaphore same as Mutex?* There is an ambiguity between binary semaphore and mutex. We might think that mutex is binary semaphore. But they are not! The purpose of mutex and semaphore are different. Mutex is locking mechanism used to synchronize access to a resource. Only one person (can be a thread) can acquire the mutex. On the other hand, semaphore is signaling mechanism (“I am done, you can carry on” kind of signal).

*10 .Can a thread acquire more than one lock (Mutex)?* Yes, it is possible that a thread access more than one resource, hence the locks

*11. Can a mutex be locked more than once*? No! It can’t as it has only one state (locked/unlocked) is associated with it. However, a recursive mutex(RLOCK) can be locked more than once, in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

*12. What will happen if a non-recursive mutex is locked more than once?* Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex.

*13. What are synchronization primitives?*

The semantics of mutex, semaphore, event, critical section, etc… are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details.

**14.6.3 Synchronization between threads**

As you see, the output of these three threads is interleaved. Because the print function is not synchronize properly. As run() function is not synchronized and run use the shared STD-OUTPUT, the output is interleaved. Thus the body of the run function is a critical section, in the sence that one and only one function should execute this body at a time – we should not allow more than one thread to execute this part at same time. To do this we can create a LOCK using the statement MY\_LOCK = threading.Lock(). A thread must ask for the lock using MY\_LOCK.acquire() before entering the critical and release the LOCK after executing the critical section.

The modified code is as below:

import threading

import time

**MY\_LOCK = threading.Lock()# getting a LOCK Here**

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, threadName,sleeptime):

"Will take <threadId,name and sleeptime>"

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.threadName = threadName

self.sleeptime = sleeptime

def run(self):

**MY\_LOCK.acquire() # ACQUIRE A LOCK before entering this function**

print "Starting " + self.threadName +'\n'

count = 0

while count < 3:

time.sleep(self.sleeptime)

count += 1

print "%s: %s : Counting %d\n" % ( self.threadName, time.ctime(time.time()),count )

print "Exiting " + self.threadName +'\n'

**MY\_LOCK.release()**

# Create new threads

thread1 = myThread(1, "Thread-1",2)

thread2 = myThread(2, "Thread-2",4)

thread3 = myThread(3, "Thread-3",6)

#craeatng a thread list

threadlist=[]

threadlist.append(thread1);

threadlist.append(thread2);

threadlist.append(thread3);

# Start new Threads

thread1.start()

thread2.start()

thread3.start()

#main thread shoudl wait for all thread to join.

for t in threadlist:

t.join()

print "Exiting Main Thread"

Output is as below:

Starting Thread-1

Thread-1: Sun Dec 08 14:25:31 2013 : Counting 1

Thread-1: Sun Dec 08 14:25:33 2013 : Counting 2

Thread-1: Sun Dec 08 14:25:35 2013 : Counting 3

Exiting Thread-1

Starting Thread-2

Thread-2: Sun Dec 08 14:25:39 2013 : Counting 1

Thread-2: Sun Dec 08 14:25:43 2013 : Counting 2

Thread-2: Sun Dec 08 14:25:47 2013 : Counting 3

Exiting Thread-2

Starting Thread-3

Thread-3: Sun Dec 08 14:25:53 2013 : Counting 1

Thread-3: Sun Dec 08 14:25:59 2013 : Counting 2

Thread-3: Sun Dec 08 14:26:05 2013 : Counting 3

Exiting Thread-3

Exiting Main Thread

**14.6.4 with Lock:**

We can use “with” statement indicating the following block executes under lock condition, for example

With MY\_LOCK:

Stmt1

Stmt2

Stmt3

Is same as:

MY\_LOCK.acquire()

Stmt1

Stmt2

Stmt3

MY\_LOCK.release()

The following code has some effect as before

def run(self):

with MY\_LOCK:

print "Starting " + self.threadName +'\n'

count = 0

while count < 3:

time.sleep(self.sleeptime)

count += 1

print "%s: %s”

print "Exiting " + self.threadName +'\n'

**14.6.3 Synchronization between threads: Rlock**

As we discussed before, Locks have 2 states: locked and unlocked. 2 methods are used to manipulate them: acquire() and release(). Those are the rules:

* if the state is unlocked: a call to acquire() changes the state to locked.
* if the state is locked: a call to acquire() blocks until another thread calls release().
* if the state is unlocked: a call to release() raises a RuntimeError exception.
* if the state is locked: a call to release() changes the state to unlocked().

RLock is a reentrant lock. acquire() can be called multiple times by the same thread without blocking. Keep in mind that release() needs to be called the same number of times to unlock the resource.

Using Lock, the second call to acquire() by the same thread will block:

MY\_LOCK = threading.Lock()

. . . . .

def run(self):

**MY\_LOCK.acquire()**

**MY\_LOCK.acquire() # Will Block Here and No output**

# Other code

MY\_LOCK.release()

MY\_LOCK.acquire()

If you use RLock, the second call to acquire() won’t block.

MY\_LOCK = threading.RLock()

def run(self):

**MY\_LOCK.acquire()**

**MY\_LOCK.acquire() # Will Work Fine as we use RLOCK**

# Other Code

MY\_LOCK.release()

MY\_LOCK.acquire()

Now question is why we need double locking or is it possible to have a double locking situation. The ans is Yes! Please go through the following example and see why double locking is required.

**Example 1:**

class X:

def \_\_init\_\_(self):

self.a = 1

self.b = 2

self.lock = threading.RLock()

def changeA(self): # Lock While Changing a

with self.lock:

self.a = self.a + 1

def changeB(self): # Lock while Changing b

with self.lock:

self.b = self.b + self.a

def changeAandB(self):

# you can use chanceA and changeB threadsave!

with self.lock:

self.changeA() # a usual lock would block in here

self.changeB()

In the above example, We can think as changeA and chnageB thread safe, we can call self.changeA() and self.changeB() without having any lock. But the intension should be like, do changeA and ChangeB automatic. Hence it is required double lock.

**Example 2:** In recursion it is obvious.

lock = threading.RLock()

def a(...):

with lock:

a(...) # somewhere inside

**14.6.3 Synchronization between threads on condition:**

**14.7 Requests : A better way to serf internet using python**

Python’s standard urllib2 module provides most of the HTTP capabilities you need, but the API is thoroughly broken. It was built for a different time — and a different web. It requires an enormous amount of work (even method overrides) to perform the simplest of tasks**.** Requests help a better way to access webpage. Requests was developed with a few PEP 20 idioms in mind.

* Beautiful is better than ugly.
* Explicit is better than implicit.
* Simple is better than complex.
* Complex is better than complicated.
* Readability counts.

14.1 Installation:

There are number of way to install as below:

1. with pip: $ pip install requests
2. with easy\_install: $ easy\_install requests
3. With pyPi: $ pip install -i <http://simple.crate.io/requests>
4. From Code: git clone git://github.com/kennethreitz/requests.git and python setup.py install

*14.2. Making a Web requests:* Making a request with Requests is very simple.

Begin by importing the Requests module and Now, let’s try to get a webpage.

>>> import requests

>>> r = requests.get('https://github.com/timeline.json')

Now, we have a Response object called r. We can get all the information we need from this object.

We can also do all kind of HTTP request using requests in very easy way.

>>> r = requests.post("http://httpbin.org/post")

>>> r = requests.put("http://httpbin.org/put")

>>> r = requests.delete("http://httpbin.org/delete")

>>> r = requests.head("http://httpbin.org/get")

>>> r = requests.options("http://httpbin.org/get")

We can also pass parameter in url as below:

>>> payload = {'key1': 'value1', 'key2': 'value2'}

>>> r = requests.get("http://httpbin.org/get", params=payload)

>>> print(r.url)

http://httpbin.org/get?key2=value2&key1=value1

Similar way we can also pass HTTP POST perameters as below:

>>> payload = {'key1': 'value1', 'key2': 'value2'}

>>> r = requests.post("http://httpbin.org/post", data=payload)

>>> print r.text #Typically, you want to send some form-encoded data — much like an HTML form.

{

...

"form": {

"key2": "value2",

"key1": "value1"

},

...

}

We can also send request with custom HTTP Headers to mimic web browser:

>>> import json

>>> url = 'https://api.github.com/some/endpoint'

>>> payload = {'some': 'data'}

>>> headers = {'content-type': 'application/json'}

>>> r = requests.post(url, data=json.dumps(payload), headers=headers)

We can also upload a file using request as below:

>>> url = 'http://httpbin.org/post'

>>> files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=files)

>>> r.text

{

...

"files": {

"file": "<censored...binary...data>"

},

...

}

Sending a request with Cookies information:

>>> url = 'http://httpbin.org/cookies'

>>> cookies = dict(cookies\_are='working')

>>> r = requests.get(url, cookies=cookies)

>>> r.text

'{"cookies": {"cookies\_are": "working"}}'

*14.3 Handling with responses:* In previous section we see the different way to send a HTTP request. After doing http request, we get a http response object and we can do following operation with the response objects:

1. Check status code:

>>> r = requests.get('http://httpbin.org/get')

>>> r.status\_code

200

2. Bad request exception

>>> bad\_r = requests.get('http://httpbin.org/status/404')

>>> bad\_r.status\_code

404

>>> bad\_r.raise\_for\_status()

Traceback (most recent call last):

File "requests/models.py", line 832, in raise\_for\_status

raise http\_error

requests.exceptions.HTTPError: 404 Client Error

>>> r = requests.get('http://httpbin.org/get')

>>> r.status\_code

200

>>> bad\_r.raise\_for\_status()

None

3. Read response header:

>>> r.headers

{

'content-encoding': 'gzip',

'transfer-encoding': 'chunked',

'connection': 'close',

'server': 'nginx/1.0.4',

'x-runtime': '148ms',

'etag': '"e1ca502697e5c9317743dc078f67693f"',

'content-type': 'application/json'

}

>>> r.headers['Content-Type']

'application/json'

>>> r.headers.get('content-type')

'application/json'

4. Getting Cookie value:

>>> url = 'http://example.com/some/cookie/setting/url'

>>> r = requests.get(url)

>>> r.cookies['example\_cookie\_name']

'example\_cookie\_value'

5. read HTML contents or Binary contents:

>>> import requests

>>> r = requests.get('https://github.com/timeline.json')

>>> r.text

u'[{"repository":{"open\_issues":0,"url":"https://github.com/...

>>> r.content # For binary data

b'[{"repository":{"open\_issues":0,"url":"https://github.com/...

>>> from PIL import Image

>>> from StringIO import StringIO

>>> i = Image.open(StringIO(r.content))

########## Get data in raw binary stremas.

>>> r.raw

<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>

>>> r.raw.read(10) # read 10 bytes.

'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'

14.4 . Authentication and persistence connection

Start with creating session:

s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')

r = s.get("http://httpbin.org/cookies")

print(r.text)

# '{"cookies": {"sessioncookie": "123456789"}}'

A complete example is as below:

TBD

14.7 Beautiful soup: A better way to extract HTML Data

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.

Here we will illustrate an example to explain how we can use this to parse the data:

Let take the following HTML code as our example:

>>> html ="""

<html>

<head><title> hello python</title></head>

<body>

<div class="c1" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c4">hello</h1></a></li>

</div>

<div class="c2" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a></li>

</div>

<div class="c3" id="i1" style="width:200px">

<li id="i1"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i3"><a href="www.dd1.com">dd3<h1 class="c5" id="i2">hello</h1></a></li>

</div>

</body>

</html>

"""

1. Let’s Print this HTML code in a butiful manner

>>> from bs4 import BeautifulSoup

>>> soup = BeautifulSoup(html)

>>> print soup.prettify()

<html>

<head>

<title>

hello python

</title>

</head>

<body>

<div class="c1" id="i1" style="width:200px">

<li id="i2">

<a href="www.dd3.com">

dd1

</a>

</li>

<li id="i2">

<a href="www.dd2.com">

dd2

</a>

</li>

<li id="i2">

<a href="www.dd1.com">

dd3

<h1 class="c4">

hello

</h1>

</a>

</li>

<div>

<div class="c2" id="i1" style="width:200px">

<li id="i2">

<a href="www.dd3.com">

dd1

</a>

</li>

<li id="i2">

<a href="www.dd2.com">

dd2

</a>

</li>

<li id="i2">

<a href="www.dd1.com">

dd3

<h1 class="c5">

hello

</h1>

</a>

</li>

<div>

<div class="c3" id="i1" style="width:200px">

<li id="i2">

<a href="www.dd3.com">

dd1

</a>

</li>

<li id="i2">

<a href="www.dd2.com">

dd2

</a>

</li>

<li id="i2">

<a href="www.dd1.com">

dd3

<h1 class="c5" id="i2">

hello

</h1>

</a>

</li>

</div>

</body>

</html>

2. accessing any html tag its name and contents.

>>> soup.title => <title> hello python</title>

>>> soup.title.name => u'title'

>>> soup.title.string => u' hello python'

>>> soup.title.parent => <head><title> hello python</title></head>

>>> soup.title.parent.name => u'head'

>>> soup.title.parent.stringv => u' hello python'

**>>> soup.div => return First DIV**

<div class="c1" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c4">hello</h1></a></li>

</div>

**>>> soup.div['class']**

[u'c1']

>>>

**>>> soup.find\_all('div') => return all DIV**

[<div class="c1" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c4">hello</h1></a></li>

</div>, <div class="c2" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a></li>

</div>, <div class="c3" id="i1" style="width:200px">

<li id="i2"><a href="www.dd3.com">dd1</a></li>

<li id="i2"><a href="www.dd2.com">dd2</a></li>

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c5" id="i2">hello</h1></a></li>

</div>]

**>>> soup.find\_all('a') => return all a**

[<a href="www.dd3.com">dd1</a>, <a href="www.dd2.com">dd2</a>, <a href="www.dd1.com">dd3<h1 class="c4">hello</h1></a>, <a href="www.dd3.com">dd1</a>, <a href="www.dd2.com">dd2</a>, <a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a>, <a href="www.dd3.com">dd1</a>, <a href="www.dd2.com">dd2</a>, <a href="www.dd1.com">dd3<h1 class="c5" id="i2">hello</h1></a>]

>>>

**>>> soup.div.find(id="i2") => Find i2 id of first DIV**

<li id="i2"><a href="www.dd3.com">dd1</a></li>

**>>> >>> soup.find('div',{'class':'c2'}) => Find DIV having class c2**

**>>> soup.find('div',{'class':'c2'}).find('li',{'id':"i2"}) => Find li having id i2 inside div having class c2**

<li id="i2"><a href="www.dd3.com">dd1</a></li>

**>>>**

>>> soup.find('div',{'class':'c2'}).find('li',{'id':"i2"}).a => Find a

<a href="www.dd3.com">dd1</a>

>>> soup.find('div',{'class':'c2'}).find('li',{'id':"i2"}).a.string => Find String init

u'dd1'

>>> soup.find('div',{'class':'c2'}).find('li',{'id':"i2"}).a['href'] =>Find Href

u'www.dd3.com'

>>>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})

[<li id="i2"><a href="www.dd3.com">dd1</a></li>, <li id="i2"><a href="www.dd2.com">dd2</a></li>, <li id="i2"><a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a></li>]

>>>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[2]

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a></li>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1]

<li id="i2"><a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a></li>

>>>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a

<a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.html

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a

<a href="www.dd1.com">dd3<h1 class="c5">hello</h1></a>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.get\_text()

u'dd3hello'

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.h1

<h1 class="c5">hello</h1>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.h1.strings

<generator object \_all\_strings at 0x02BA0F58>

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.h1.string

u'hello'

>>> soup.find('div',{'class':'c2'}).find\_all('li',{'id':"i2"})[-1].a.string

>>>

3. Accessing All HTML tags class