Scrapy Project Proposal

(*Support for Different robots.txt Parsers)*

***Organisation**:- Python Software Foundation*
***Sub–org**:- Scrapy*

***Student Information*:-**

1) **Name**:- Maram Sumanth

2) **Github username**:- maramsumanth

3) **IRC Nickname**:- maramsumanth

4) **Email**:- maram.sumanth@gmail.com

5) **Mobile Number**:- +91 9121359358

6) **Time Zone**:- IST (UTC+5:30)

7) **Mentors**:- Nikita, Konstantin

***My contributions to scrapy and its dependencies*:-**

1) scrapy/scrapy:-

https://github.com/scrapy/scrapy/pull/3687
https://github.com/scrapy/scrapy/pull/3579
https://github.com/scrapy/scrapy/pull/3643
https://github.com/scrapy/scrapy/pull/3638

2) scrapy/w3lib:-

https://github.com/scrapy/w3lib/pull/119

***Project Info*:-**

Project Title:

Support for Different robots.txt Parsers

Brief Explanation:

Scrapy’s existing robots.txt parser is not fully compliant, but the more compliant alternatives are difficult to package and use within Scrapy’s pure-python development tree. This project would introduce a new interface for parsers of robots.txt files, so that a better parser may be substituted as required or desired.

Goal of this project:-

Ancient parsers follow the standard rule which accounts for limited directives, but modern smart parsers supports robots.txt files with more directives/extensions from the real world. This project aims to implement interface for scrapy and then improve cpython to account for other directives. If time permits during this GSoC period, we can develop a smarter parser for scrapy.

Brief Info:-

Internally Scrapy uses urllib.robotparser which follows the original standard. But, the modern robots.txt parsers does include many extensions. There are many standards that have been evolved like MK1994, MK1996 and GYM2008. The first two comprise the traditional robots.txt standard and the last describes some extensions. GYM2008 is shorthand for the robots.txt syntax extensions (path wildcards, the Crawl-delay directive and the Sitemap directive) agreed upon by Google, Yahoo and Microsoft in 2008.

The Crawl-delay and Sitemap directives are ignored by older parsers and are a useful addition to the standard.

Why interface?:-

Consider the following simple code which returns whether a given user_agent is allowed to fetch the URL in robots.txt file or not:-

For urllib.robotparser : rp.can_fetch(user_agent, url)
For reppy : rp.allowed(url, user_agent)

Instead of having to remember different method names for different parsers which does the same work, it is better to have a single method name for any parser, which the a developer can use with ease.

After implementation : rp.is_allowed(url, user_agent)
of this Interface

Implementation of interface:-

During discussion, I have proposed **ROBOTSTXT_PARSER** variable in settings which permits the user to change the parser according to his need.

First of all we need to add abstract interface in the downloadermiddleware's robotstxt.py. This is achieved by first importing parsers.py file in robotstxt middleware in downloadermiddlewares and then:-

Consider the following line in the robotstxt.py middleware:-

rp = robotparser.RobotFileParser(response.url)

By replacing above line with,

rp = parsers.StdlibParser(response.url)
(or)
rp = parsers.ReppyParser(response.url)
(or)
Any Parser (If any can be implemented in future)

according to the setting **ROBOTSTXT_PARSER** (which is stdlib Parser by default), where parsers.py file consists of:-

1) BaseRobotFileParser class
2) StdlibParser class (imports BaseRobotFileParser class)
3) ReppyParser class (imports BaseRobotFileParser class)
(For future) 4) abcParser class (imports BaseRobotFileParser class)

Abstract Methods to be defined in the Base class:-

1) parse
2) read
3) is_allowed
4) sitemaps
5) get_crawl_delay
6) get_request_rate

*****and other methods (can be added by including other directives)

Description of base robotstxt parser class:-

class BaseRobotFileParser:

"Base parser for robots.txt files, this class is inherited by all the parser interfaces defined"

def __init__(self, url):

"Constructor"

@abstractmethods
def is_allowed(self, url, user_agent):

"Returns a bool whether the user_agent is permitted to visit the URL or not"

def read(self, url):

"Reads the robots.txt URL and feeds it to parser"

def sitemaps(self):

"Returns the sitemap URLs present in the robots.txt file. If nothing is present returns None"

def parse(self, content):

"Parses the contents of robots.txt file"

def get_crawl_delay(self, user_agent):

"Returns the Crawl-delay value for the given user-agent"

def get_request_rate(self, user_agent):

"Returns the request-rate value for the given user-agent"

- Method names aren't fixed, which can be changed accordingly after discussing with mentors.

Implementation of interface for a parser (assume XYZParser) :-

class XYZParser(BaseRobotFileParser):

def __init__(self, url):

BaseRobotFileParser.__init__(self, url)

self.rp = "Define here where this XYZParser is coded"

"For example
self.rp = robotparser.RobotFileParser()
self.rp.set_url(url) **for** stdlib python parser"

"self.rp = Robots.fetch(url) **for** reppy parser"

def is_allowed(self, url, user_agent):

"Sets the given URL"
return self.rp.can_fetch(user_agent, url) **for** urllib.robotparser

return self.rp.allowed(url, user_agent) **for** reppy parser

def sitemaps(self):

return self.rp.sitemaps for reppy parser

return None for stdlib parser
"As urllib.robotparser does not support sitemaps directive. (I could improve in cpython by adding this directive)"

def get_crawl_delay(self, user_agent):

return self.rp.crawl_delay(user_agent) **for** urllib.robotparser

return robots.agent(user_agent).delay **for** reppy

def get_request_rate(self, user_agent):

return self.rp.request_rate(user_agent) for urllib.robotparser

- Here, I have shown example considering reppy and stdlib parser and their corresponding methods supported, remaining methods can be implemented based on the type of parser we choose because some directives supported by one parser may not be supported by another.

Example showing the implementation:-

from parsers import abcParser

rp = parsers.abcParser(url)
"Above statement is nothing but calling the implemented robotstxt parser interface"

Now we can use the methods as defined above with rp, using syntax
"rp.method(url, user_agent)"

For Example:-

```
rp.parse(lines)
rp.is_allowed(url, user_agent)
rp.get_crawl_delay(user_agent)
rp.get_request_rate(user_agent)
.
.
…. etc
```

- So, any parser interface which is implemented in future (presently in this GSoC project we are implementing Interfaces for two existing parsers), should inherit base class (BaseRobotFileParser) and its methods.

Comparison of reppy parser v/s stdlib python parser (urllib.robotparser):-

Case (I):- Order of Allow/Disallow directives

Ex:- For https://google.com/robots.txt. Consider the following group in the robots.txt file:-

```
User-agent: *
Disallow: /search
Allow: /search/about
```

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("https://google.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch('*','https://google.com/search/about')
False
```

```
>>> from reppy.robots import Robots
>>> robots = Robots.fetch('http://google.com/robots.txt')
>>> agent = robots.agent('*')
>>> agent.allowed('https://google.com/search/about')
True
```

It is evident from the above two images that both contradict and stdlib python parser is showing incorrect. This is so because urllib.robotparser follows the ordering standard (To take the first matching directive). This would have been correct if the directive order is reversed ------- **To be implemented (Task 1)**.

- Since the reppy parser and Google's implementation follows that Allow patterns with equal or more path components in the directive path win over a matching Disallow pattern. Task 1 can be implemented by first splitting ('/') the path of Allow/Disallow directives, and then sorting it based on length.

Case (II):- Behaviour towards wildcards

Ex:- Consider https://google.com/robots.txt. I have chosen this example, since it has directives containing wildcards.
For the following group in the robots.txt:-

User-agent: *
Allow: /?hl=
Disallow: /?hl=*&

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("https://google.com/robots.txt")
>>> rp.read()
>>> rp.can_fetch('*','https://google.com/hl=')
False
>>> rp.can_fetch('*','https://google.com/hl=&')
False
```

```
>>> from reppy.robots import Robots
>>> robots = Robots.fetch('http://google.com/robots.txt')
>>> agent = robots.agent('*')
>>> agent.allowed('https://google.com/?hl=')
True
>>> agent.allowed('https://google.com/?hl=&')
False
```

As you can see there is issue with the robotparser on wildcards. Modern smart parsers does account for wildcards, and we can implement/improve in stdlib python parser such that it supports wildcards --------- **To be implemented (Task 2)**.

Case (III):- Support for Sitemaps

Note that Crawl-delay and Sitemap directives are specific to GYM2008 syntax, but implementing them here is not a problem. They are just additional information.

The present stdlib parser supports for user-agent, disallow, allow, crawl-delay and request-rate. We can also add one more line checking for sitemaps, which is user unspecific and proceed accordingly ------- **To be implemented (Task 3)**.

- Some of the lesser known directives which are implemented in neither and can be considered for improvement (**After discussing with mentors**) are:-
  (Visit-time, Host, Indexpage, Clean-param)

Case (IV):- Order of precedence for rules with wildcards

This case is indirectly a combination of I and II cases, which needs to be explicitly checked for better result. Consider the following directives for a particular user agent in robots.txt file.

User-agent: *
Allow: /page
Disallow: /*.html

Can **https://example.com/page.html** be crawled by '*' useragent?

Well, this creates a lot of confusion since Google's Implementation considers that the verdict is undefined. From the above two lines, we can clearly understand the intention of the website owner, i.e the URL shouldn't be crawled.

Since scrapy declares itself as a polite crawler by obeying robots.txt. To be polite in all cases, we need to consider this improvement.

Bugs in stdlib parser as described in rerp, which needs to be solved:-

- When a robots.txt file contains a BOM (Byte Order Mark), present cpython robotparser doesn't make any accommodation for them.

  When the file consists of,
  [BOM] User-agent : *
  the user agent line is seen as garbage and this bug can have significant consequences.

- There is a bug in handling of paths that contain a %-encoded forward slash, MK1996 says that they shouldn't be translated but the robotparser module does.

New features to be added in cpython robotparser:-

- Support for is_expired() and expiration_date which is obtained by looking for an HTTP Expires header when fetching robots.txt, if it is present then storing it in expiration_date otherwise default to present_time + 1 week (MK1996 standard)

- Adding **line_count** attribute to indicate the size of robots.txt file, and useragent_count (which gives number of user agents) and directive_count (for user agent unspecific directives)
  Ex:- Sitemaps_count (user agent unspecific)
  …...etc

- Support for Non-ASCII characters in robots.txt file. By decoding the file with the encoding specified in the HTTP Content-type headersen with robots.txt file. If no encoding is specified, it defaults to ISO-8859-1 as per the HTTP specifications (**After discussing with mentors**)

***Proposed Timeline*:-**

Accepted students proposal announced (Until May 6):-

1) Get confident with scrapy project by submitting more pull requests, fixing bugs and helping developers in issue tracker. These contributions will help me in getting accustomed to scrapy code which could make easier my future tasks

2) If organisation or mentors want any modifications to be done in my proposal, implementing them accordingly

Community Bonding Period (May 7 – May 26):-

1) Work on the proposed implementation, get feedback from mentors and discuss with them

2) Get familiar with the seomoz/reppy code and how it is implemented, which helps in designing a better parser

3) Interacting with mentors regarding how to report, develop each task and review my progress

4) Keep submitting PR's to repository, which boosts my confidence

Implementing the Interface (*May 27 – June 4):-*

1) Understanding the robotstxt.py middleware and modifying it to incorporate the new setting and building the necessary code as described above

2) During this time, I will define some methods which accounts for the directives, also some of the lesser known directives* (described above) to code in the Interface

June 5 – June 13:-

1) Implementing the parsers module by coding the BaseRobotFileParser class, which is inherited by any parser

2) Add tests and docs

June 14 – June 21:-

1) Start writing code for the first parser interface i.e stdlib parser, with the help of urllib.robotparser

2) Add tests and docs

June 22 – July 12:-

1) Exploit all the existing functionalities of the reppy parser and understanding it, which helps in building a better custom parser (If time permits during GSoC period)

2) Also looking at the other python parsers which might be useful (Ex:- rerp), and adding interface for them also if necessary

3) Start writing code for the second parser interface i.e reppy parser, with the help of reppy

4) Test the written module extensively

July 13 – July 20:-

1) Discuss with the mentors and finalize the interface, if necessary do the required modifications like implementing support for lesser known directives. Debug the written code and add tests

Improving cpython robotparser (July 21 – July 29):-

1) Task 2, solving bugs and adding features as described above

2) Writing documentation to the till now written code

July 30 – Aug 12:-

1) Task 1 + Task 3 + Case IV

2) Adding tests, docs and improve code coverage to the above three tasks

Aug 13 – Aug 19:-

1) Finalization of project by discussing with mentors. Fix bugs and improve code structure

2) Debug the written code by hosting a robots.txt file (which contain sitemaps, Allow/Disallow directives with wildcards) locally, file content is extracted from robots.txt file parser and testing/verifying the output

Submit Final Project (Aug 20 – Aug 26):-

1) Wrap up, complete the project and submit the final report. If necessary format the code, update docs and tests

- Whenever I get any free time during any week, I will write tests (which improves accuracy), docs, implement small features and properly format the code.

Other Commitments:-

My vacations are from 10th May to 11th July. So my classes commence from 12th July, and my mid semester exams may be during last week of August, which does not significantly affect the project.

I have been actively contributing to scrapy from the past four months by solving bugs, adding features and corresponding docs as evident from the above PR's. I applied only to scrapy, I haven't applied to any other organisation in GSoC 2019.

Extra Information:-

Drive link to Resume:-
https://drive.google.com/file/d/1OTEWU6T3ayFhmfDAo1Phh3TquvQ26_fF/view?usp=sharing

University:- Indian Institute of Technology, Roorkee

Major:- Electronics and Communications Engineering, B-Tech

Alternate email id:- maram.sumanth4@gmail.com

I hope that I will be able to complete this GSoC Summer Project within the stipulated time by taking suggestions/improvements from the mentors and implementing them accordingly.