Scrapy : Support for different robots.txt parsers

- Name: Anubhav Dinesh Patel
- Email: anubhavp28@gmail.com
- Github: anubhavp28
- University: Indian Institute of Information Technology Kalyani
- Course: Bachelor of Technology in Computer Science and Engineering
- Course Term: 2017/18 - 2020/21 (4 Year)
- Timezone: IST (GMT +5:30)
- Phone: +91 9082079014 (India)
- Resume: https://anubhavp28.github.io/Resume/resume.pdf

Proposal

This project is about introducing a new interface for robots.txt parsers in scrapy, allowing users of scrapy to substitute a new robots.txt parser if they do not want to use the inbuilt one.

Deliverables

At the end of this project, I hope to deliver:

- A new interface for scrapy to communicate with any robots.txt parser. Parsers will have to implement this interface to allow integration with scrapy.
- A modified version of the existing inbuilt parser that works with the new interface.
- An implementation of this interface on top of existing 3rd party parsers - reppy and Robotexclusionrulesparser (rerp).
- **(Stretch Goal)** A pure python robots.txt parser.

Implementation

The implementation language will be Python. The implementation of any class/method will accompany tests and documentation for the same.

Details

1. **Interface for robots.txt parsers**

- A new setting `ROBOTSTXT_PARSER` will be added which will point to the python import path of a class implementing the new interface. This class will be used by scrapy to communicate with the robots.txt parser.
- An abstract class `BaseRobotsTxtParser` will be introduced. This abstract class will act as a guide to the interface spec. A user who wish to substitute a new parser will have to extend this class and implement its methods.

  Here is the description of how the `BaseRobotsTxtParser` class will look like:

  `class` **`BaseRobotsTxtParser`**

  - `__init__(robotstxt_url, user_agent)`
    - Parameters :
      - robotstxt_url : URL of robots.txt file
      - user_agent : user-agent to use while retrieving rules.
  - `parse(robotstxt_body=None)`

    Calls the robots.txt parser to parse the contents of robots.txt file.

    - Parameters :
      - robotstxt_body : **(Optional)** A string containing the contents of robots.txt file. If None, `robotstxt_url` will be used to fetch robots.txt for parsing.
  - `allowed(url, user_agent=None)`

    Returns whether the given url is allowed or not.

    - Parameters :
      - url : url to check permission for
      - user_agent : **(Optional)** *user-agent* to use while checking**.** If None, *user-agent* given during the instantiation will be used.
  - `sitemaps()`

    Returns a list of links to the sitemaps on the website. Sitemaps directive is not user-agent specific. If there is no sitemaps specified in `robots.txt`, return `None`.

- `crawl_delay(user_agent=None)`

  Returns the value of the `Crawl-delay` parameter from `robots.txt` for the given *user-agent*. If there is no such parameter or it doesn't apply to the *user-agent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

  - Parameters :

    - user_agent: **(Optional)** *user-agent* to find `Crawl-delay` for. If `None`, *user-agent* specified during instantiation will be used.

- `preferred_host()`

  Returns preferred domain specified with Host directive. If nothing is specified, returns `None`.

Here is a sample implementation of this interface on top of python inbuilt robots.txt parser `urllib.robotparser`:

```
class InbuiltRobotsTxtParser(BaseRobotsTxtParser):

    def __init__(self, robotstxt_url, user_agent):
        self.robotstxt_url = robotstxt_url
        self.user_agent = user_agent
        self.rp = urllib.robotparser.RobotFileParser()
        self.rp.set_url(robotstxt_url)

    def parse(self, robotstxt_body=None):
        if robotstxt_body:
            self.rp.parse(robotstxt_body)
        else:
            # fetch using url provided and parse
            # though, we shouldn't use a synchronous fetch
            self.rp.read()

    def allowed(self, url, user_agent=None):
        user_agent = user_agent or self.user_agent
        return self.rp.can_fetch(user_agent, url)

    def sitemaps(self):
        """urllib.robotparser does not support sitemap directive"""
        return None
```

```
def crawl_delay(self, user_agent=None):
    user_agent = user_agent or self.user_agent
    return self.rp.crawl_delay(user_agent)

def preferred_host()
    """urllib.robotparser does not support host directive"""
    return None
```

- Two new classes will be introduced - `ReppyRobotsParser` and `RerpRobotsParser` both implementing the new interface on top of reppy and Robotexclusionrulesparser (rerp) respectively. Thus, scrapy will have support for three different robots.txt parsers out of the box.
- Modify `RobotsTxtMiddleware` to use `ROBOTSTXT_PARSER` to instantiate the parser (similar to how `HttpCacheMiddleware` uses `HTTPCACHE_STORAGE` to instantiate the cache storage backend)
- Modify `RobotsTxtMiddleware` to use the new interface.

Discussion

- **How to handle not fully compliant parsers**
  The user is free to substitute any parser. Hence, it can easily be the case that a substituted parser may not support all of them. While implementing interface on top such parsers, how should we handle the case of a parser not supporting a certain directive. For example, Host directive may not be supported by all parsers (`urllib.robotparser` does not support host directive) and the interface specification I proposed recommends keeping a preferred_host() function, then should calling preferred_host() raise some custom exception notifying that the parser doesn't support it or should it just act like as if there is no preferred domain is specified with host directive in robots.txt?

- **Scrapy does not use crawl-delay value**
  Since this project will provide fully compliant robots.txt parser support to Scrapy. Maybe, we can somehow use `crawl-delay` value to automatically set Scrapy's `DONWLOAD_DELAY` setting. Hence, making Scrapy respect `crawl-delay` by default?

- **Should we leave the behaviour when the server response (while retrieving robots.txt) is something other than success (HTTP 2XX status code) to the parser to implement?**
  Different robots tend to behave differently, for example Yandex bot simply assumes unlimited access if the server response is something other than success. While google follow a complex set of rules described here - https://developers.google.com/search/reference/robots_txt.

  If we do decide to go with this, we can change the parser interface to accept `scrapy.http.Response` object. Maybe something like this:

```
def __init__(self, response, user_agent):
        self.response = response
        self.user_agent = user_agent
        self.rp = urllib.robotparser.RobotFileParser()
        self.rp.set_url(response.url)

def parse(self):
        self.rp.parse(self.response.body)
```

2. Pure Python robots.txt parser

Discussion

Conflicting specification exists for robots.txt, therefore a detailed discussion is necessary before commencing implementation of our own parser. Below, I have listed a set of topics that we can discuss during the community bonding period.

- **Priority among Allow and Disallow rules**
  The allow directive is now widely supported by parsers and robots. The 1997 Internet Draft of robots.txt specification recommends that the first matching robots.txt pattern should decide whether the URL is allowed or not. If no match is found, the default assumption should be that the URL is allowed. The Python's in-built `RobotFileParser` uses this approach.

  Google's implementation differs in that Allow patterns with equal or more characters in the directive path win over a matching Disallow pattern. The `reppy` parser uses this approach.

- **Behaviour when the server response is something other than success (HTTP 2XX status code)**

I think, we should leave it to the parsers to implement behaviour when the server response is something other than success (HTTP 2XX status code). This way we get a cleaner interface, otherwise any middleware would first need to check for original server response and then make a query with a call to `allowed` function to check if the URL is okay for crawling.

Different robots tend to behave differently, for example Yandex bot simply assumes unlimited access if the server response is something other than success.

For our own parser, we can follow the 1997 Internet Draft of robots.txt specification that recommends

- When the server response indicates that the resource doesn't exist (HTTP status code 404), assume that the access to the website is not restricted.
- When server response indicating access restrictions (HTTP status code 401 or 403), assume the access to the website is completely restricted.
- When server response indicates redirection (HTTP status code 3XX), follow the redirect until a resource can be found. Though it doesn't specify any upper bound on the redirects.

- **Allow and Disallow directives without parameters**

```
Disallow:
Allow:
```

Different robots and parsers, tend to interpret `Disallow` and `Allow` directives different when they are without any parameters.

When `Disallow` directive is without any parameter, Googlebot simply ignores the directive. While Yandex and rerp, assume that unlimited access is provided (nothing is disallowed).

When `Allow` directive is without any parameter, Googlebot and Yandex simply ignores the directive. While rerp, assume that nothing is allowed (access is completely restricted).

- *** and $ wildcards**
  A lot of robots support * and $ wildcards. `reppy` supports theses wildcards.
  - * matches any sequence of characters
  - $ matches the end of the URL.

  Example (disallow all files with .asp extension) :

  ```
  User-agent: Googlebot
  Disallow: /*.asp$
  ```

  Google's specifications also support * wildcard in user-agent (such `user-agent: googlebot*`)

- **Support for other directives**
  Other directives such as `Clear-param`, `Visit-time` and `Request-rate`, can also be supported.

- **Making our parser configurable**
  Since, conflicting specifications exist, the results can vary considerably when using a different specification. Maybe, we can introduce settings that will allow the user to control the behaviour of our parser.

Implementation

- I have provided pseudo-code for a robots.txt parser. While writing this pseudocode I have taken liberty and have skipped certain details, that I would definitely keep in mind while writing the code during GSoC. I have below a non-exhaustive list of such details that I have skipped while writing the pseudo-code.

  > Specifications recommend a case insensitive substring match without version information for comparing user-agents. I have went with direct string comparison in the pseudocode.

  > For comparing urls, I have used simple prefix match in pseudocode - no support for wildcards.

```
class RobotsParserPython:

        Instance Variables
```

- rules: A dictionary mapping user-agents to a dictionary containing the rules for them.
- currentDirective: current directive
- userAgentBuffer: list of user-agent current record is for

Methods

```
function parse(robotstxt_content):

      lines = robotstxt_content.split('\n')

      for line in lines:

         1. Remove the comment part of the line (ie, part of the
            line starting from `#`)

         2. Split line using `:` as delimiter

         3. If split produces does not produce exactly 2 strings,
            stop processing this line further and move to the next
            line. This could be a line that only contain comment.

         4. Label the 2 strings produced as field and value.

         5. field = tolower(field) # Since the field name is case
            insensitive, convert field to lowercase.

         6. value = trim(value)

         7. If field name is not among the known directives, stop
            processing this line further and move to the next
            line. Robots.txt specification allows for lines with
            fields not explicitly specified by the specification,
            for future extension of the format.

         8. call _handleDirective with field name and value


function _handleDirectives(field, value):

   1. previousDirective = self.currentDirective

   2. self.currentDirective = field

   3. switch (field) :

         a. case “user-agent”:
```

- If previousDirective != “user-agent”:
  - # beginning of a new record
  - self.userAgentBuffer = [] # empty list
- # multiple user-agents can correspond to this record. Store all of them in userAgentBuffer.
- userAgent = tolower(value)
- self.userAgentBuffer.append(userAgent)
- self.rules[userAgent] = {} # new dict

b. case “disallow”:

- foreach userAgent in self.userAgentBuffer:
  - self.rules[userAgent][‘disallow’].append(value)

c. case “crawl-delay”:

- foreach userAgent in self.userAgentBuffer:
  - self.rules[userAgent][‘crawl-delay’] = value

d. case “sitemap”:

- # sitemap directive is not user-agent specific
- self.rules[‘*’][‘sitemap’].append(value)

e. case “host”:

- # host directive is not user-agent specific
- self.rules[‘*’][‘host’] = value

f. default:

- foreach userAgent in self.userAgentBuffer:
  - self.rules[userAgent][field].append(value)

Apply length rule (similar to reppy and Google’s implementation) to order allow & disallow directives for a user-agent

function orderDirectives(rules):

1. orderedDirectives = []
2. allowRules = rules[‘allow’]
3. disallowRules = rules[‘disallow’]
4. directives = []
5. foreach rule in allowRules:
    a. directives.append({“directive”: “allow”, “rule”:rule})
6. foreach rule in disallowRules:
    a. directive.append({“directive”:”disallow”, “rule”:rule})
7. sort(directives, in descending order based on string length of rule part)
8. return directives

function isAllowed(url, userAgent):

1. #convert https://www.example.com/test/me to /test/me
2. url = getRelativeURL(url) # easy to implement hence, pseudocode not given.
3. orderedDirectives = orderDirectives(self.rules[userAgent]) # sort rules based on length rule
4. if no rules found for userAgent: #orderedDirectives is empty
    a. orderedDirectives = orderDirectives(self.rules[‘*’])
5. allowed = True
6. foreach directive in orderedDirectives:
    - If (url.beginsWith(directives[‘rule’]):
        - if (directive[‘directive’]==’allow’)
            - break
        - else:
            - allowed=False
            - break
7. return allowed

```
function getSitemaps():
   1. return self.rules[‘*’][‘sitemap’]
```

- For implementing support for * and $ wildcards, we can use regular expression. We can escape metacharacters apart from * and $, and replace * with .* to give it the correct meaning (with respect to robots.txt spec). With limited testing that I have done, it seems to produce the correct result.

```
  def parser_match(pattern, url):
      s = re.split(r'([$*])', pattern)
      for index, substr in enumerate(s):
          if substr not in ['*', '$']:
              s[index] = re.escape(substr)
          else:
              s[index] = s[index].replace('*', '.*')
      pattern = ''.join(s)
      if re.findall(pattern, url):
          return True
      return False

>>> parser_match(r'/test/*.php$', '/test/me.pph')
False
>>> parser_match(r'/test/*.php$', '/test/me.phpp')
False
>>> parser_match(r'/test/*.php$', '/test/me.php')
True
>>> parser_match(r'/test/*.php$', '/test/')
False
```

Code Contribution

Pull requests

- https://github.com/scrapy/scrapy/pull/3660
- https://github.com/scrapy/scrapy/pull/3689
- https://github.com/scrapy/scrapy/pull/3692
- https://github.com/scrapy/scrapy/pull/3738
- https://github.com/scrapy/scrapy/pull/3662
- https://github.com/scrapy/scrapy/pull/3735
- https://github.com/scrapy/scrapy/pull/3737

Issues

- https://github.com/scrapy/scrapy/issues/3683
- https://github.com/scrapy/scrapy/issues/3664
- https://github.com/scrapy/scrapy/issues/3661
- https://github.com/scrapy/scrapy/issues/3736
- https://github.com/scrapy/scrapy/issues/3734

Timeline

Community Bonding Period : May 6, 2019 - May 26, 2019

- Setup development environment.
- Actively participate in regular meetings.
- Finalise interface for robots.txt parsers.
- Commence discussion on topics listed in the proposal above.
- Finalise deadlines and milestones.
- Increase familiarity with community practices and processes.
- Contribute bug fixes/patches.
- Setup blog for GSoC.

Week 1 : May 27, 2019 - June 2, 2019

- Finalise interface for robots.txt parser.
- Implement abstract class `BaseRobotsTxtParser`
- Implement the new interface on top of Python's inbuilt `urllib.robotparser`

Week 2 : June 3, 2019 - June 9, 2019

- Modify `RobotsTxtMiddleware` class to use the new interface rather than directly calling `RobotFileParser`
- Document the new interface and `ROBOTSTXT_PARSER` setting.

- Write tests.

Week 3 : June 10, 2019 - June 16, 2019

- Document the new interface and write tests.
- Implement the new interface on top of `reppy`.

Week 4 : June 17, 2019 - June 23, 2019

- Document the new interface and write tests.
- Implement the new interface on top of `rerp`.

Week 5 : June 24, 2019 - June 30, 2019

- < Buffer Week > Complete leftover work and discussion with mentors.
- Discussion about implementation of a pure python robots.txt parser.

Week 6 : June 18, 2019 - June 25, 2019

- Discussion about implementation of a pure python robots.txt parser.
- Implement basic functionality related to parsing a robots.txt file.
- Write documentation and tests.

Week 7 : July 1, 2019 - July 7, 2019

- Implement basic functionality related to parsing a robots.txt file.
- Implement ability to check whether a URL is allowed to crawl or not.
- Write documentation and tests.

Week 8 : July 8, 2019 - July 14, 2019

- Implement ability to check whether a URL is allowed to crawl or not.
- Implement ability to retrieve a list of sitemap, the preferred host and crawl-delay.
- Write documentation and tests.

Week 9 : July 15, 2019 - July 21, 2019

- Implement support for * and $ wildcards in the new parser.
- Write documentation and tests.

Week 10 : July 28, 2019 - Aug 4, 2019

- Write documentation and tests.

- Discuss with mentors how the parser can be extended further.

Week 11 : Aug 5, 2019 - Aug 11, 2019

- Discuss with mentors how the parser can be extended further. Maybe, Implement support for uncommon directives.

Week 12 : Aug 12, 2019 - Aug 18, 2019

- < Buffer Week > Complete leftover work and discussion with mentors.

Final Project Submission Week: Aug 19, 2019 - Aug 26, 2019

- Wrap up, complete leftover work and final report.

Other Commitments

- I have only applied to Scrapy in GSoC 2019.
- I will have my semester ending examination in May. I don't have exact dates yet, though it is likely to be starting from May 10. The examination period will be of 5 days. It won't impact my productivity a lot.