icrawler Documentation

Release 0.6.6

Kai Chen

Jun 27, 2023

Contents

CHAPTER 1

icrawler

1.1 Introduction

Documentation: http://icrawler.readthedocs.io/

Try it with `pip install icrawler` or `conda install -c hellock icrawler`.

This package is a mini framework of web crawlers. With modularization design, it is easy to use and extend. It supports media data like images and videos very well, and can also be applied to texts and other type of files. Scrapy is heavy and powerful, while icrawler is tiny and flexible.

With this package, you can write a multiple thread crawler easily by focusing on the contents you want to crawl, keeping away from troublesome problems like exception handling, thread scheduling and communication.

It also provides built-in crawlers for popular image sites like **Flickr** and search engines such as **Google**, **Bing** and **Baidu**. (Thank all the contributors and pull requests are always welcome!)

1.2 Requirements

Python 3.5+ (recommended).

1.3 Examples

Using built-in crawlers is very simple. A minimal example is shown as follows.

```
from icrawler.builtin import GoogleImageCrawler

google_crawler = GoogleImageCrawler(storage={'root_dir': 'your_image_dir'})
google_crawler.crawl(keyword='cat', max_num=100)
```

You can also configurate number of threads and apply advanced search options. (Note: compatible with 0.6.0 and later versions)

```
from icrawler.builtin import GoogleImageCrawler

google_crawler = GoogleImageCrawler(
    feeder_threads=1,
    parser_threads=2,
    downloader_threads=4,
    storage={'root_dir': 'your_image_dir'})
filters = dict(
    size='large',
    color='orange',
    license='commercial,modify',
    date=((2017, 1, 1), (2017, 11, 30)))
google_crawler.crawl(keyword='cat', filters=filters, max_num=1000, file_idx_offset=0)
```

For more advanced usage about built-in crawlers, please refer to the documentation.

Writing your own crawlers with this framework is also convenient, see the tutorials.

1.4 Architecture

A crawler consists of 3 main components (Feeder, Parser and Downloader), they are connected with each other with FIFO queues. The workflow is shown in the following figure.

- `url_queue` stores the url of pages which may contain images
- `task_queue` stores the image url as well as any meta data you like, each element in the queue is a dictionary and must contain the field `img_url`
- Feeder puts page urls to `url_queue`
- Parser requests and parses the page, then extracts the image urls and puts them into `task_queue`
- Downloader gets tasks from `task_queue` and requests the images, then saves them in the given path.

Feeder, parser and downloader are all thread pools, so you can specify the number of threads they use.

CHAPTER 2

Documentation index

2.1 Installation

The quick way (with pip):

```
pip install icrawler
```

or (with conda)

```
conda install -c hellock icrawler
```

You can also manually install it by

```
git clone git@github.com:hellock/icrawler.git
cd icrawler
python setup.py install
```

If you fail to install it on Linux, it is probably caused by *lxml*. See here for solutions.

2.2 Built-in crawlers

This framework contains 6 built-in image crawlers.

- Google
- Bing
- Baidu
- Flickr
- General greedy crawl (crawl all the images from a website)
- UrlList (crawl all images given an url list)

2.2.1 Search engine crawlers

The search engine crawlers (Google, Bing, Baidu) have universal APIs. Here is an example of how to use the built-in crawlers.

```
from icrawler.builtin import BaiduImageCrawler, BingImageCrawler, GoogleImageCrawler

google_crawler = GoogleImageCrawler(
    feeder_threads=1,
    parser_threads=1,
    downloader_threads=4,
    storage={'root_dir': 'your_image_dir'})
filters = dict(
    size='large',
    color='orange',
    license='commercial,modify',
    date=((2017, 1, 1), (2017, 11, 30)))
google_crawler.crawl(keyword='cat', filters=filters, offset=0, max_num=1000,
                     min_size=(200,200), max_size=None, file_idx_offset=0)

bing_crawler = BingImageCrawler(downloader_threads=4,
                                storage={'root_dir': 'your_image_dir'})
bing_crawler.crawl(keyword='cat', filters=None, offset=0, max_num=1000)

baidu_crawler = BaiduImageCrawler(storage={'root_dir': 'your_image_dir'})
baidu_crawler.crawl(keyword='cat', offset=0, max_num=1000,
                    min_size=(200,200), max_size=None)
```

The filter options provided by Google, Bing and Baidu are different. Supported filter options and possible values are listed below.

GoogleImageCrawler:

- `type` – "photo", "face", "clipart", "linedrawing", "animated".
- `color` – "color", "blackandwhite", "transparent", "red", "orange", "yellow", "green", "teal", "blue", "purple", "pink", "white", "gray", "black", "brown".
- `size` – "large", "medium", "icon", or larger than a given size (e.g. ">640x480"), or exactly is a given size ("=1024x768").
- `license` – "noncommercial"(labeled for noncommercial reuse), "commercial"(labeled for reuse), "noncommercial,modify"(labeled for noncommercial reuse with modification), "commercial,modify"(labeled for reuse with modification).
- `date` – "pastday", "pastweek" or a tuple of dates, e.g. `((2016, 1, 1), (2017, 1, 1))` or `((2016, 1, 1), None)`.

BingImageCrawler:

- `type` – "photo", "clipart", "linedrawing", "transparent", "animated".
- `color` – "color", "blackandwhite", "red", "orange", "yellow", "green", "teal", "blue", "purple", "pink", "white", "gray", "black", "brown"
- `size` – "large", "medium", "small" or larger than a given size (e.g. ">640x480").
- `license` – "creativecommons", "publicdomain", "noncommercial", "commercial", "noncommercial,modify", "commercial,modify".
- `layout` – "square", "wide", "tall".

- `people` – "face", "portrait".
- `date` – "pastday", "pastweek", "pastmonth", "pastyear".

BaiduImageCrawler:

- `type`: "portrait", "face", "clipart", "linedrawing", "animated", "static"
- `color`: "red", "orange", "yellow", "green", "purple", "pink", "teal", "blue", "brown", "white", "black", "blackandwhite".

When using `GoogleImageCrawler`, language can be specified via the argument `language`, e.g., `google_crawler.crawl(keyword='cat', language="us")`.

Note: Tips: Search engines will limit the number of returned images, even when we use a browser to view the result page. The limitation is usually 1000 for many search engines such as google and bing. To crawl more than 1000 images with a single keyword, we can specify different date ranges.

```
google_crawler.crawl(
    keyword='cat',
    filters={'date': ((2016, 1, 1), (2016, 6, 30))},
    max_num=1000,
    file_idx_offset=0)
google_crawler.crawl(
    keyword='cat',
    filters={'date': ((2016, 6, 30), (2016, 12, 31))},
    max_num=1000,
    file_idx_offset='auto')
# set `file_idx_offset` to "auto" so that filenames can be consecutive numbers (e.g.,␣
↪1001 ~ 2000)
```

2.2.2 Flickr crawler

```
from datetime import date
from icrawler.builtin import FlickrImageCrawler

flickr_crawler = FlickrImageCrawler('your_apikey',
                                    storage={'root_dir': 'your_image_dir'})
flickr_crawler.crawl(max_num=1000, tags='child,baby',
                     group_id='68012010@N00', min_upload_date=date(2015, 5, 1))
```

Supported optional searching arguments are listed in https://www.flickr.com/services/api/flickr.photos.search.html. Here are some examples.

- `user_id` – The NSID of the user who's photo to search.
- `tags` – A comma-delimited list of tags.
- `tag_mode` – Either "any" for an OR combination of tags, or "all" for an AND combination.
- `text` – A free text search. Photos who's title, description or tags contain the text will be returned.
- `min_upload_date` – Minimum upload date. The date can be in the form of `datetime.date` object, an unix timestamp or a string.
- `max_upload_date` – Maximum upload date. Same form as `min_upload_date`.
- `group_id` – The id of a group who's pool to search.

- `extras` – A comma-delimited list of extra information to fetch for each returned record. See here for more details.
- `per_page` – Number of photos to return per page.

Some advanced searching arguments, which are not updated in the Flickr API, are also supported. Valid arguments and values are shown as follows.

- `color_codes` – A comma-delimited list of color codes, which filters the results by your chosen color(s). Please see any Flickr search page for the corresponding relations between the colors and the codes.
- `styles` – A comma-delimited list of styles, including `blackandwhite`, `depthoffield`, `minimalism` and `pattern`.
- `orientation` – A comma-delimited list of image orientation. It can be `landscape`, `portrait`, `square` and `panorama`. The default includes all of them.

Another parameter `size_preference` is available for Flickr crawler, it define the preferred order of image sizes. Valid values are shown as follows.

- original
- large 2048: 2048 on longest side†
- large 1600: 1600 on longest side†
- large: 1024 on longest side*
- medium 800: 800 on longest side†
- medium 640: 640 on longest side
- medium: 500 on longest side
- small 320: 320 on longest side
- small: 240 on longest side
- thumbnail: 100 on longest side
- large square: 150x150
- square: 75x75

`size_preference` can be either a list or a string, if not specified, all sizes are acceptable and larger sizes are prior to smaller ones.

Note: * Before May 25th 2010 large photos only exist for very large original images. † Medium 800, large 1600, and large 2048 photos only exist after March 1st 2012.

2.2.3 Greedy crawler

If you just want to crawl all the images from some website, then `GreedyImageCrawler` may be helpful.

```
from icrawler.builtin import GreedyImageCrawler

greedy_crawler = GreedyImageCrawler(storage={'root_dir': 'your_image_dir'})
greedy_crawler.crawl(domains='http://www.bbc.com/news', max_num=0,
                     min_size=None, max_size=None)
```

The argument `domains` can be either an url string or list.

2.2.4 URL list crawler

If you have already got an image url list somehow and want to download all images using multiple threads, then `UrlListCrawler` may be helpful.

```
from icrawler.builtin import UrlListCrawler

urllist_crawler = UrlListCrawler(downloader_threads=4,
                                 storage={'root_dir': 'your_image_dir'})
urllist_crawler.crawl('url_list.txt')
```

You can see the complete example in *test.py*, to run it

```
python test.py [options]
```

`options` can be `google`, `bing` , `baidu`, `flickr`, `greedy`, `urllist` or `all`, using `all` by default if no arguments are specified. Note that you have to provide your flickr apikey if you want to test FlickrCrawler.

2.3 Extend and write your own

It is easy to extend `icrawler` and use it to crawl other websites. The simplest way is to override some methods of Feeder, Parser and Downloader class.

1. **Feeder**

   The method you need to override is

   ```
   feeder.feed(self, **kwargs)
   ```

   If you want to offer the start urls at one time, for example from 'http://example.com/page_url/1' up to 'http://example.com/page_url/10'

   ```
   from icrawler import Feeder

   class MyFeeder(Feeder):
       def feed(self):
           for i in range(10):
               url = 'http://example.com/page_url/{}'.format(i + 1)
               self.output(url)
   ```

2. **Parser**

   The method you need to override is

   ```
   parser.parse(self, response, **kwargs)
   ```

   `response` is the page content of the url from `url_queue`, what you need to do is to parse the page and extract file urls, and then put them into `task_queue`. Beautiful Soup package is recommended for parsing html pages. Taking `GoogleParser` for example,

   ```
   class GoogleParser(Parser):

       def parse(self, response):
           soup = BeautifulSoup(response.content, 'lxml')
           image_divs = soup.find_all('div', class_='rg_di rg_el ivg-i')
           for div in image_divs:
   ```

(continues on next page)

(continued from previous page)

```
            meta = json.loads(div.text)
            if 'ou' in meta:
                yield dict(file_url=meta['ou'])
```

3. **Downloader**

   If you just want to change the filename of downloaded images, you can override the method

```
downloader.get_filename(self, task, default_ext)
```

   The default names of downloaded files are increasing numbers, from 000001 to 999999.

   Here is an example of using other filename formats instead of numbers as filenames.

```
import base64

from icrawler import ImageDownloader
from icrawler.builtin import GoogleImageCrawler
from six.moves.urllib.parse import urlparse


class PrefixNameDownloader(ImageDownloader):

    def get_filename(self, task, default_ext):
        filename = super(PrefixNameDownloader, self).get_filename(
            task, default_ext)
        return 'prefix_' + filename


class Base64NameDownloader(ImageDownloader):

    def get_filename(self, task, default_ext):
        url_path = urlparse(task['file_url'])[2]
        if '.' in url_path:
            extension = url_path.split('.')[-1]
            if extension.lower() not in [
                    'jpg', 'jpeg', 'png', 'bmp', 'tiff', 'gif', 'ppm', 'pgm'
            ]:
                extension = default_ext
        else:
            extension = default_ext
        # works for python 3
        filename = base64.b64encode(url_path.encode()).decode()
        return '{}.{}'.format(filename, extension)


google_crawler = GoogleImageCrawler(
    downloader_cls=PrefixNameDownloader,
    # downloader_cls=Base64NameDownloader,
    downloader_threads=4,
    storage={'root_dir': 'images/google'})
google_crawler.crawl('tesla', max_num=10)
```

   If you want to process meta data, for example save some annotations of the images, you can override the method

```
downloader.process_meta(self, task):
```

   Note that your parser need to put meta data as well as file urls into `task_queue`.

If you want to do more with the downloader, you can also override the method

```
downloader.download(self, task, default_ext, timeout=5, max_retry=3,
                    overwrite=False, **kwargs)
```

You can retrieve tasks from `task_queue` and then do what you want to do.

4. **Crawler**

   You can either use the base class `Crawler` or inherit from it. Two main apis are

```
crawler.__init__(self, feeder_cls=Feeder, parser_cls=Parser,
                 downloader_cls=Downloader, feeder_threads=1,
                 parser_threads=1, downloader_threads=1,
                 storage={'backend': 'FileSystem', 'root_dir': 'images'},
                 log_level=logging.INFO)
```

   and

```
crawler.crawl(self, feeder_kwargs={}, parser_kwargs={}, downloader_kwargs={})
```

   So you can use your crawler like this

```
crawler = Crawler(feeder_cls=MyFeeder, parser_cls=MyParser,
                  downloader_cls=ImageDownloader, downloader_threads=4,
                  storage={'backend': 'FileSystem', 'root_dir': 'images'})
crawler.crawl(feeder_kwargs=dict(arg1='blabla', arg2=0),
              downloader_kwargs=dict(max_num=1000, min_size=None))
```

   Or define a class to avoid using complex and ugly dictionaries as arguments.

```
class MyCrawler(Crawler):

    def __init__(self, *args, **kwargs):
        super(GoogleImageCrawler, self).__init__(
            feeder_cls=MyFeeder,
            parser_cls=MyParser,
            downloader_cls=ImageDownloader,
            *args,
            **kwargs)

    def crawl(self, arg1, arg2, max_num=1000, min_size=None, max_size=None, file_
↪idx_offset=0):
        feeder_kwargs = dict(arg1=arg1, arg2=arg2)
        downloader_kwargs = dict(max_num=max_num,
                                 min_size=None,
                                 max_size=None,
                                 file_idx_offset=file_idx_offset)
        super(MyCrawler, self).crawl(feeder_kwargs=feeder_kwargs,
                                     downloader_kwargs=downloader_kwargs)

crawler = MyCrawler(downloader_threads=4,
                    storage={'backend': 'FileSystem', 'root_dir': 'images'})
crawler.crawl(arg1='blabla', arg2=0, max_num=1000, max_size=(1000,800))
```

2.4 How to use proxies

A powerful `ProxyPool` class is provided to handle the proxies. You will need to override the `Crawler.set_proxy_pool()` method to use it.

If you just need a few (for example less than 30) proxies, you can override it like the following.

```
def set_proxy_pool(self):
    self.proxy_pool = ProxyPool()
    self.proxy_pool.default_scan(region='overseas', expected_num=10,
                                 out_file='proxies.json')
```

Then it will scan 10 valid overseas (out of mainland China) proxies and automatically use these proxies to request pages and images.

If you have special requirements on proxies, you can use ProxyScanner and write your own scan functions to satisfy your demands.

```
def set_proxy_pool(self):
    proxy_scanner = ProxyScanner()
    proxy_scanner.register_func(proxy_scanner.scan_file,
                                {'src_file': 'proxy_overseas.json'})
    proxy_scanner.register_func(your_own_scan_func,
                                {'arg1': '', 'arg2': ''})
    self.proxy_pool.scan(proxy_scanner, expected_num=10, out_file='proxies.json')
```

Every time when making a new request, a proxy will be selected from the pool. Each proxy has a weight from 0.0 to 1.0, if a proxy has a greater weight, it has more chance to be selected for a request. The weight is increased or decreased automatically according to the rate of successful connection.

2.5 API reference

2.5.1 crawler

Crawler base class

class icrawler.crawler.**Crawler**(*feeder_cls=<class 'icrawler.feeder.Feeder'>*, *parser_cls=<class 'icrawler.parser.Parser'>*, *downloader_cls=<class 'icrawler.downloader.Downloader'>*, *feeder_threads=1*, *parser_threads=1*, *downloader_threads=1*, *storage={'backend': 'FileSystem', 'root_dir': 'images'}*, *log_level=20*, *extra_feeder_args=None*, *extra_parser_args=None*, *extra_downloader_args=None*)

Base class for crawlers

session
A Session object.

Type *Session*

feeder
A Feeder object.

Type *Feeder*

parser
A Parser object.

Type *Parser*

downloader
A Downloader object.

Type *Downloader*

signal
A Signal object shared by all components, used for communication among threads

Type *Signal*

logger
A Logger object used for logging

Type Logger

crawl(*feeder_kwargs=None*, *parser_kwargs=None*, *downloader_kwargs=None*)
Start crawling

This method will start feeder, parser and download and wait until all threads exit.

Parameters

- **feeder_kwargs** (*dict, optional*) – Arguments to be passed to `feeder.start()`
- **parser_kwargs** (*dict, optional*) – Arguments to be passed to `parser.start()`
- **downloader_kwargs** (*dict, optional*) – Arguments to be passed to `downloader.start()`

init_signal()
Init signal

3 signals are added: `feeder_exited`, `parser_exited` and `reach_max_num`.

set_logger(*log_level=20*)
Configure the logger with log_level.

set_proxy_pool(*pool=None*)
Construct a proxy pool

By default no proxy is used.

Parameters **pool** (`ProxyPool, optional`) – a `ProxyPool` object

set_session(*headers=None*)
Init session with default or custom headers

Parameters **headers** – A dict of headers (default None, thus using the default header to init the session)

set_storage(*storage*)
Set storage backend for downloader

For full list of storage backend supported, please see `storage`.

Parameters **storage** (*dict or* `BaseStorage`) – storage backend configuration or instance

2.5.2 feeder

class icrawler.feeder.**Feeder**(*thread_num*, *signal*, *session*)
Bases: icrawler.utils.thread_pool.ThreadPool

Base class for feeder.

A thread pool of feeder threads, in charge of feeding urls to parsers.

thread_num
An integer indicating the number of threads.

Type int

global_signal
A Signal object for communication among all threads.

Type *Signal*

out_queue
A queue connected with parsers' inputs, storing page urls.

Type Queue

session
A session object.

Type *Session*

logger
A logging.Logger object used for logging.

Type Logger

workers
A list storing all the threading.Thread objects of the feeder.

Type list

lock
A Lock instance shared by all feeder threads.

Type Lock

feed(***kwargs*)
Feed urls.

This method should be implemented by users.

worker_exec(***kwargs*)
Target function of workers

class icrawler.feeder.**SimpleSEFeeder**(*thread_num*, *signal*, *session*)
Bases: *icrawler.feeder.Feeder*

Simple search engine like Feeder

feed(*url_template*, *keyword*, *offset*, *max_num*, *page_step*)
Feed urls once

Parameters

- **url_template** – A string with parameters replaced with "{}".
- **keyword** – A string indicating the searching keyword.
- **offset** – An integer indicating the starting index.

- **max_num** – An integer indicating the max number of images to be crawled.
- **page_step** – An integer added to offset after each iteration.

class icrawler.feeder.**UrlListFeeder**(*thread_num*, *signal*, *session*)
Bases: *icrawler.feeder.Feeder*

Url list feeder which feed a list of urls

feed(*url_list*, *offset=0*, *max_num=0*)
Feed urls.

This method should be implemented by users.

2.5.3 parser

class icrawler.parser.**Parser**(*thread_num*, *signal*, *session*)
Bases: icrawler.utils.thread_pool.ThreadPool

Base class for parser.

A thread pool of parser threads, in charge of downloading and parsing pages, extracting file urls and put them into the input queue of downloader.

global_signal
A Signal object for cross-module communication.

session
A requests.Session object.

logger
A logging.Logger object used for logging.

threads
A list storing all the threading.Thread objects of the parser.

thread_num
An integer indicating the number of threads.

lock
A threading.Lock object.

parse(*response*, ***kwargs*)
Parse a page and extract image urls, then put it into task_queue.

This method should be overridden by users.

Example

```
>>> task = {}
>>> self.output(task)
```

worker_exec(*queue_timeout=2*, *req_timeout=5*, *max_retry=3*, ***kwargs*)
Target method of workers.

Firstly download the page and then call the *parse()* method. A parser thread will exit in either of the following cases:

1. All feeder threads have exited and the url_queue is empty.
2. Downloaded image number has reached required number.

Parameters

- **queue_timeout** (*int*) – Timeout of getting urls from `url_queue`.
- **req_timeout** (*int*) – Timeout of making requests for downloading pages.
- **max_retry** (*int*) – Max retry times if the request fails.
- ****kwargs** – Arguments to be passed to the *parse()* method.

2.5.4 downloader

class icrawler.downloader.**Downloader**(*thread_num*, *signal*, *session*, *storage*)

Bases: icrawler.utils.thread_pool.ThreadPool

Base class for downloader.

A thread pool of downloader threads, in charge of downloading files and saving them in the corresponding paths.

task_queue
A queue storing image downloading tasks, connecting `Parser` and *Downloader*.

Type *CachedQueue*

signal
A Signal object shared by all components.

Type *Signal*

session
A session object.

Type *Session*

logger
A logging.Logger object used for logging.

workers
A list of downloader threads.

Type list

thread_num
The number of downloader threads.

Type int

lock
A threading.Lock object.

Type Lock

storage
storage backend.

Type *BaseStorage*

clear_status()
Reset fetched_num to 0.

download(*task*, *default_ext*, *timeout=5*, *max_retry=3*, *overwrite=False*, ***kwargs*)
Download the image and save it to the corresponding path.

Parameters

- **task** (*dict*) – The task dict got from `task_queue`.

- **timeout** (*int*) – Timeout of making requests for downloading images.
- **max_retry** (*int*) – the max retry times if the request fails.
- ****kwargs** – reserved arguments for overriding.

get_filename(*task*, *default_ext*)
Set the path where the image will be saved.

The default strategy is to use an increasing 6-digit number as the filename. You can override this method if you want to set custom naming rules. The file extension is kept if it can be obtained from the url, otherwise `default_ext` is used as extension.

Parameters **task** (*dict*) – The task dict got from `task_queue`.

Output: Filename with extension.

process_meta(*task*)
Process some meta data of the images.

This method should be overridden by users if wanting to do more things other than just downloading the image, such as saving annotations.

Parameters **task** (*dict*) – The task dict got from task_queue. This method will make use of fields other than `file_url` in the dict.

reach_max_num()
Check if downloaded images reached max num.

Returns if downloaded images reached max num.

Return type bool

set_file_idx_offset(*file_idx_offset=0*)
Set offset of file index.

Parameters **file_idx_offset** – It can be either an integer or 'auto'. If set to an integer, the filename will start from `file_idx_offset` + 1. If set to `'auto'`, the filename will start from existing max file index plus 1.

worker_exec(*max_num*, *default_ext=''*, *queue_timeout=5*, *req_timeout=5*, ***kwargs*)
Target method of workers.

Get task from `task_queue` and then download files and process meta data. A downloader thread will exit in either of the following cases:

1. All parser threads have exited and the task_queue is empty.
2. Downloaded image number has reached required number(max_num).

Parameters

- **queue_timeout** (*int*) – Timeout of getting tasks from `task_queue`.
- **req_timeout** (*int*) – Timeout of making requests for downloading pages.
- ****kwargs** – Arguments passed to the *download()* method.

class icrawler.downloader.**ImageDownloader**(*thread_num*, *signal*, *session*, *storage*)
Bases: *icrawler.downloader.Downloader*

Downloader specified for images.

get_filename(*task*, *default_ext*)
Set the path where the image will be saved.

The default strategy is to use an increasing 6-digit number as the filename. You can override this method if you want to set custom naming rules. The file extension is kept if it can be obtained from the url, otherwise `default_ext` is used as extension.

Parameters **task** (*dict*) – The task dict got from `task_queue`.

Output: Filename with extension.

keep_file(*task*, *response*, *min_size=None*, *max_size=None*)
Decide whether to keep the image

Compare image size with `min_size` and `max_size` to decide.

Parameters

- **response** (*Response*) – response of requests.
- **min_size** (*tuple or None*) – minimum size of required images.
- **max_size** (*tuple or None*) – maximum size of required images.

Returns whether to keep the image.

Return type bool

worker_exec(*max_num*, *default_ext='jpg'*, *queue_timeout=5*, *req_timeout=5*, ***kwargs*)
Target method of workers.

Get task from `task_queue` and then download files and process meta data. A downloader thread will exit in either of the following cases:

1. All parser threads have exited and the task_queue is empty.
2. Downloaded image number has reached required number(max_num).

Parameters

- **queue_timeout** (*int*) – Timeout of getting tasks from `task_queue`.
- **req_timeout** (*int*) – Timeout of making requests for downloading pages.
- ****kwargs** – Arguments passed to the `download()` method.

2.5.5 storage

class icrawler.storage.**BaseStorage**
Bases: `object`

Base class of backend storage

exists(*id*)
Check the existence of some data

Parameters **id** (*str*) – unique id of the data in the storage

Returns whether the data exists

Return type bool

max_file_idx()
Get the max existing file index

Returns the max index

Return type int

write(*id*, *data*)
Abstract interface of writing data

Parameters

- **id** (*str*) – unique id of the data in the storage.
- **data** (*bytes or str*) – data to be stored.

class icrawler.storage.**FileSystem**(*root_dir*)
Bases: icrawler.storage.base.BaseStorage

Use filesystem as storage backend.

The id is filename and data is stored as text files or binary files.

exists(*id*)
Check the existence of some data

Parameters **id** (*str*) – unique id of the data in the storage

Returns whether the data exists

Return type bool

max_file_idx()
Get the max existing file index

Returns the max index

Return type int

write(*id*, *data*)
Abstract interface of writing data

Parameters

- **id** (*str*) – unique id of the data in the storage.
- **data** (*bytes or str*) – data to be stored.

class icrawler.storage.**GoogleStorage**(*root_dir*)
Bases: icrawler.storage.base.BaseStorage

Google Storage backend.

The id is filename and data is stored as text files or binary files. The root_dir is the bucket address such as gs://<your_bucket>/<your_directory>.

exists(*id*)
Check the existence of some data

Parameters **id** (*str*) – unique id of the data in the storage

Returns whether the data exists

Return type bool

max_file_idx()
Get the max existing file index

Returns the max index

Return type int

write(*id*, *data*)
Abstract interface of writing data

Parameters

- **id** (*str*) – unique id of the data in the storage.
- **data** (*bytes or str*) – data to be stored.

2.5.6 utils

class icrawler.utils.**CachedQueue**(**args*, ***kwargs*)
Bases: Queue.Queue, object

Queue with cache

This queue is used in *ThreadPool*, it enables parser and downloader to check if the page url or the task has been seen or processed before.

_cache
cache, elements are stored as keys of it.

Type OrderedDict

cache_capacity
maximum size of cache.

Type int

is_duplicated(*item*)
Check whether the item has been in the cache

If the item has not been seen before, then hash it and put it into the cache, otherwise indicates the item is duplicated. When the cache size exceeds capacity, discard the earliest items in the cache.

Parameters **item** (*object*) – The item to be checked and stored in cache. It must be immutable or a list/dict.

Returns Whether the item has been in cache.

Return type bool

put(*item*, *block=True*, *timeout=None*, *dup_callback=None*)
Put an item to queue if it is not duplicated.

put_nowait(*item*, *dup_callback=None*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

class icrawler.utils.**Proxy**(*addr=None*, *protocol='http'*, *weight=1.0*, *last_checked=None*)
Bases: object

Proxy class

addr
A string with IP and port, for example '123.123.123.123:8080'

Type str

protocol
'http' or 'https'

Type str

weight
A float point number indicating the probability of being selected, the weight is based on the connection time and stability

Type float

last_checked
A UNIX timestamp indicating when the proxy was checked

Type time

format()
Return the proxy compatible with requests.Session parameters

Returns A dict like {'http': '123.123.123.123:8080'}

Return type dict

to_dict()
convert detailed proxy info into a dict

Returns

A dict with four keys: addr, protocol, `weight` and `last_checked`

Return type dict

class icrawler.utils.**ProxyPool**(*filename=None*)
Bases: `object`

Proxy pool class

ProxyPool provides friendly apis to manage proxies.

idx
Index for http proxy list and https proxy list.

Type dict

test_url
A dict containing two urls, when testing if a proxy is valid, test_url['http'] and test_url['https'] will be used according to the protocol.

Type dict

proxies
All the http and https proxies.

Type dict

addr_list
Address of proxies.

Type dict

dec_ratio
When decreasing the weight of some proxy, its weight is multiplied with *dec_ratio*.

Type float

inc_ratio
Similar to *dec_ratio* but used for increasing weights, default the reciprocal of *dec_ratio*.

Type float

weight_thr
The minimum weight of a valid proxy, if the weight of a proxy is lower than *weight_thr*, it will be removed.

Type float

logger
A logging.Logger object used for logging.

Type Logger

add_proxy(*proxy*)
Add a valid proxy into pool

You must call *add_proxy* method to add a proxy into pool instead of directly operate the *proxies* variable.

decrease_weight(*proxy*)
Decreasing the weight of a proxy by multiplying dec_ratio

default_scan(*region='mainland'*, *expected_num=20*, *val_thr_num=4*, *queue_timeout=3*, *val_timeout=5*, *out_file='proxies.json'*, *src_files=None*)
Default scan method, to simplify the usage of *scan* method.

It will register following scan functions: 1. scan_file 2. scan_cnproxy (if region is mainland) 3. scan_free_proxy_list (if region is overseas) 4. scan_ip84 5. scan_mimiip After scanning, all the proxy info will be saved in out_file.

Parameters

- **region** – Either 'mainland' or 'overseas'
- **expected_num** – An integer indicating the expected number of proxies, if this argument is set too great, it may take long to finish scanning process.
- **val_thr_num** – Number of threads used for validating proxies.
- **queue_timeout** – An integer indicating the timeout for getting a candidate proxy from the queue.
- **val_timeout** – An integer indicating the timeout when connecting the test url using a candidate proxy.
- **out_file** – the file name of the output file saving all the proxy info
- **src_files** – A list of file names to scan

get_next(*protocol='http'*, *format=False*, *policy='loop'*)
Get the next proxy

Parameters

- **protocol** (*str*) – 'http' or 'https'. (default 'http')
- **format** (*bool*) – Whether to format the proxy. (default False)
- **policy** (*str*) – Either 'loop' or 'random', indicating the policy of getting the next proxy. If set to 'loop', will return proxies in turn, otherwise will return a proxy randomly.

Returns

If format is true, then return the formatted proxy which is compatible with requests.Session parameters, otherwise a Proxy object.

Return type *Proxy* or dict

increase_weight(*proxy*)
Increase the weight of a proxy by multiplying inc_ratio

is_valid(*addr*, *protocol='http'*, *timeout=5*)
Check if a proxy is valid

Parameters

- **addr** – A string in the form of 'ip:port'
- **protocol** – Either 'http' or 'https', different test urls will be used according to protocol.
- **timeout** – A integer indicating the timeout of connecting the test url.

Returns

If the proxy is valid, returns {'valid': True, 'response_time': xx} otherwise returns {'valid': False, 'msg': 'xxxxxx'}.

Return type dict

load(*filename*)
Load proxies from file

proxy_num(*protocol=None*)
Get the number of proxies in the pool

Parameters **protocol** (*str, optional*) – 'http' or 'https' or None. (default None)

Returns If protocol is None, return the total number of proxies, otherwise, return the number of proxies of corresponding protocol.

remove_proxy(*proxy*)
Remove a proxy out of the pool

save(*filename*)
Save proxies to file

scan(*proxy_scanner*, *expected_num=20*, *val_thr_num=4*, *queue_timeout=3*, *val_timeout=5*, *out_file='proxies.json'*)
Scan and validate proxies

Firstly, call the *scan* method of *proxy_scanner*, then using multiple threads to validate them.

Parameters

- **proxy_scanner** – A ProxyScanner object.
- **expected_num** – Max number of valid proxies to be scanned.
- **val_thr_num** – Number of threads used for validating proxies.
- **queue_timeout** – Timeout for getting a proxy from the queue.
- **val_timeout** – An integer passed to *is_valid* as argument *timeout*.
- **out_file** – A string or None. If not None, the proxies will be saved into *out_file*.

validate(*proxy_scanner*, *expected_num=20*, *queue_timeout=3*, *val_timeout=5*)
Target function of validation threads

Parameters

- **proxy_scanner** – A ProxyScanner object.
- **expected_num** – Max number of valid proxies to be scanned.
- **queue_timeout** – Timeout for getting a proxy from the queue.
- **val_timeout** – An integer passed to *is_valid* as argument *timeout*.

class icrawler.utils.**ProxyScanner**
Proxy scanner class

ProxyScanner focuses on scanning proxy lists from different sources.

proxy_queue
The queue for storing proxies.

scan_funcs
Name of functions to be used in *scan* method.

scan_kwargs
Arguments of functions

scan_threads
A list of *threading.thread* object.

logger
A *logging.Logger* object used for logging.

is_scanning()
Return whether at least one scanning thread is alive

register_func(*func_name*, *func_kwargs*)
Register a scan function

Parameters

- **func_name** – The function name of a scan function.
- **func_kwargs** – A dict containing arguments of the scan function.

scan()
Start a thread for each registered scan function to scan proxy lists

scan_cnproxy()
Scan candidate (mainland) proxies from http://cn-proxy.com

scan_file(*src_file*)
Scan candidate proxies from an existing file

scan_free_proxy_list()
Scan candidate (overseas) proxies from http://free-proxy-list.net

scan_ip84(*region='mainland'*, *page=1*)
Scan candidate proxies from http://ip84.com

Parameters

- **region** – Either 'mainland' or 'overseas'.
- **page** – An integer indicating how many pages to be scanned.

scan_mimiip(*region='mainland'*, *page=1*)
Scan candidate proxies from http://mimiip.com

Parameters

- **region** – Either 'mainland' or 'overseas'.
- **page** – An integer indicating how many pages to be scanned.

class icrawler.utils.**Session**(*proxy_pool*)
Bases: requests.sessions.Session

get(*url*, ***kwargs*)
Sends a GET request. Returns Response object.

Parameters

- **url** – URL for the new Request object.

- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

post(*url*, *data=None*, *json=None*, ***kwargs*)
Sends a POST request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

class icrawler.utils.**Signal**
Bases: `object`

Signal class

Provides interfaces for set and get some globally shared variables(signals).

signals
A dict of all signal names and values.

init_status
The initial values of all signals.

get(*name*)
Get a signal value by its name.

Parameters **name** – a string indicating the signal name.

Returns Value of the signal or None if the name is invalid.

names()
Return all the signal names

reset()
Reset signals with their initial values

set(***signals*)
Set signals.

Parameters **signals** – A dict(key-value pairs) of all signals. For example {'signal1': True, 'signal2': 10}

class icrawler.utils.**ThreadPool**(*thread_num*, *in_queue=None*, *out_queue=None*, *name=None*)
Bases: `object`

Simple implementation of a thread pool

This is the base class of `Feeder`, `Parser` and `Downloader`, it incorporates two FIFO queues and a number of "workers", namely threads. All threads share the two queues, after each thread starts, it will watch the `in_queue`, once the queue is not empty, it will get a task from the queue and process as wanted, then it will put the output to `out_queue`.

Note: This class is not designed as a generic thread pool, but works specifically for crawler components.

---

name
thread pool name.

Type str

thread_num
number of available threads.

Type int

in_queue
input queue of tasks.

Type Queue

out_queue
output queue of finished tasks.

Type Queue

workers
a list of working threads.

Type list

lock
thread lock.

Type Lock

logger
standard python logger.

Type Logger

connect(*component*)
Connect two ThreadPools.

The `in_queue` of the second pool will be set as the `out_queue` of the current pool, thus all the output will be input to the second pool.

Parameters **component** (`ThreadPool`) – the ThreadPool to be connected.

Returns the modified second ThreadPool.

Return type *ThreadPool*

2.6 Release notes

2.6.1 0.6.1 (2018-05-25)

- **New**: Add an option to skip downloading when the file already exists.

2.6.2 0.6.0 (2018-03-17)

- **New**: Make the api of search engine crawlers (GoogleImageCrawler, BingImageCrawler, BaiduImageCrawler) universal, add the argument `filters` and remove arguments `img_type`, `img_color`, `date_min`, etc.
- **New**: Add more search options (type, color, size, layout, date, people, license) for Bing (Thanks @kirtanp).
- **New**: Add more search options (type, color, size) for Baidu.

- **Fix**: Fix the json parsing error of `BaiduImageCrawler` when some invalid escaped characters exist.

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

I

T

U

V

W