recaptcha_manager

Release 0.1.0

Charchit Agarwal

Apr 01, 2023

CONTENTS:

Average solving time for recaptchas by solving services like 2Captcha, Anticaptcha, etc. is around 30-60s at best, which is often a bottleneck for most scripts relying on them. recaptcha-manager aims to alleviate this problem by truly "managing" your recaptcha solving needs without really changing how your script functions. It uses those same services, but with a non-blocking architecture and some maths to seemingly bring that solving time down to less than a second. A brief run down of how it works is given below:

1. **Efficient, non-blocking architecture**: Conventional approaches often require your script to wait for the captcha request to be registered and completely solved by the solving service before proceeding. This is not the case with recaptcha-manager. After your script signals that it wants more recaptchas to be solved (via a quick function call), the control is returned to it immediately. This is possible because the actual communication with the captcha solving service, including registering the captcha task and requesting it's answer, happens in a background process. When recaptcha-manager receives the answer to a captcha request in this background process, it stores it in shared memory so your script can then access it at it's own leisure. Therefore, you can *manually pre-send* recaptcha requests before your program actually needs them, while it continues to do what it was doing. Then when your program actually requires the captcha answers, you may find that those recaptchas have already been solved or are about to be solved, significantly lowering the time you have to wait.
2. **The Maths**: Recaptcha-manager can collect relevant statistics including how frequently your script requires recaptchas, the service's solving speed, the number being currently solved, and many more. It then mathematically analyses these factors to accurately predict how many captchas your script will require in the near future and *automatically pre-sends* those many requests to the captcha solving service whenever you request more recaptchas to be solved. What this results in is that whenever your program actually wants a recaptcha, there will be one already solved and available. It's worth adding that this mathematical analysis is very accurate and only uses recent statistics, which makes sure that the solved captchas won't expire due to more requests than required being sent to the solving service.

Some other core features of recaptcha-manager are summarized below:

- **Quick Integration** - Supports API of popular captcha solving services like Anticaptcha, 2Captcha and CapMonster. Supports Windows, UNIX, and macOS.
- **Flexibility** - Works equally well on applications requiring 2-3 captchas a minute as well as those requiring 40+ captchas a minute
- **Adaptability** - Can readjust even if your applications' rate of requesting captchas drastically changes midway
- **Unification** - If you use multiple captcha solving services, then you can use all of them simultaneously using recaptcha-manager, or switch between them incase of an error.
- **Efficiency** - Apart from sending HTTP requests to communicate with the solving service's API in a separate background process, the requests are also sent asynchronously so that the service response times do not slow down scripts requiring a high volume of recaptchas

---

Note: This package uses multiprocessing to spawn a service process which handles captcha requests in the background. Therefore, your main code must be under a `if __name__ == "__main__"` clause (more information here) if you are running on Windows. A very simple example of how to do this is given below:

```
# Original code

def main():
    func()

def func():
    pass

# Not protected
main()
```

```
# Edited code

def main():
    func()

def func():
    pass

# Protected!
if __name__ == "__main__":
    main()
```

CHAPTER

ONE

GLOSSARY

- **Pre-sending** : Pre-sending in captcha solving context refers to when you send a variable number of captcha requests to be solved before you actually need them which helps minimize the waiting time for the future. There are two types: *automatic* and *manual*. As the name suggests, in *automatic* pre-sending, recaptcha-manager accurately handles the pre-sending for you based on a number of statistics it collects. This type of pre-sending is only supported with *AutoManagers*. On the contrary, in *manual* pre-sending the user is expected to decide exactly how many captcha requests to pre-send, and when. This is supported by both *AutoManagers* and *ManaualManagers*.
- **Solving service** : The captcha solving service(s) that you use. Currently, recaptcha-manager supports Anti-captcha,2Captcha and CapMonster.
- **Service process** : The background process that communicates with the solving service's API. These processes do not interrupt your program and are spawned using multiprocessing
- **Captcha parameters** : The set of parameters which identify the captcha you want to solve. In this context, these are the **type** of recaptcha, which can be "v2" or "v3", the **url** where the captcha was found, and finally the google **sitekey** of the captcha. All three must be a valid combination otherwise the solving service may refuse to solve them. The combination of all three identify exactly which recaptcha to solve.

CHAPTER

TWO

QUICKSTART

Integrating recaptcha-manager with your program is incredibly simple. It uses a *manager*, which you use to send and request captcha answers, and a *service process*, the background process which sends the captcha requests to the captcha solving service of your choice.

For convenience, there are full code examples provided here.

New in version 0.1.1: Support is now available for Windows platforms as well as UNIX systems and macOS.

The following sections go into more details about the capabilities of manager and service processes.

CHAPTER

THREE

CHOOSING A MANAGER

As mentioned before, managers in recaptcha-manager are objects of classes that your program uses to send and receive recaptcha requests from the captcha solving service. Internally, they do this by communicating with the service process on your behalf. There are two types of managers that can be created, `AutoManager` and `ManualManager`. While both these managers can be used to send and fetch requests from solving services, their use cases differ.

3.1 AutoManagers

`AutoManager` supports *automatic pre-sending* which can accurately reduce the waiting time to receive recaptcha answers from the solving service to a negligible amount. It relies on statistics collected overtime and stored internally to make accurate predictions to do so. However, because it uses *automatic pre-sending*, one instance of `AutoManager` can only handle one set of parameters for a recaptcha (because otherwise it would not know which parameters to automatically pre-send). For example, consider two captchas, **Captcha A** and **Captcha B** with the following parameters:

	Captcha A	Captcha B
type	v2	v2
url	www.google.com	www.gmail.com
sitekey	xxxxx	yyyyy

Solving them both requires the creation of two separate `AutoManager` instances, one for each, since both the captchas have different captcha parameters. There is no limit on the amount of `AutoManager` instances you can create.

Therefore, `AutoManager` is best suited for use cases where you need to repeatedly solve a lot of recaptchas with similar parameters, like submitting a form with a recaptcha for a website periodically, since automatic pre-sending would ensure that captcha answers are always available when you need them and you wouldn't need to create many AutoManagers either.

3.2 ManualManagers

As the name suggests, `ManualManager` gives more control to you and uses less resources at the expense of not supporting *automatic pre-sending*. Unlike `AutoManager`, a single `ManualManager` can be used to solve recaptchas with different captcha parameters. For example, consider two captchas, **Captcha A** and **Captcha B** with the following parameters:

	Captcha A	Captcha B
type	v2	v2
url	www.google.com	www.gmail.com
sitekey	xxxxx	yyyyy

Both these captchas can be solved with a single instance of ManualManager. However, this also means that `ManualManager` cannot support automatic pre-sending as it wouldn't know which captcha parameters to send when automatically pre-sending since it can solve more than one set of captcha parameters. You can, however, *manually pre-send* captchas whenever you need.

This makes `ManualManager` particularly useful for cases where automatic pre-sending is impractical. For example, if your program is scraping a lot of websites, it would only need a couple of recaptchas per website, if any. Therefore, automatic pre-sending would be useless since you would probably visit each site only once and can simply request the exact amount of recaptchas you need with `ManualManager` for any site, whenever you wish. Additionally, to save time, you can use manual pre-sending here instead. For instance, you can ask for the recaptcha to be solved for a particular site before you do some other time intensive task (like loading the website if you are rendering while scraping). Then, when you are done and actually require the captcha, it may have already been solved.

CHAPTER

FOUR

USING A MANAGER

This section goes into detail about all the managers and their supported functions. Keep in mind that any code examples that follow are only snippets. Check here for full code examples

4.1 AutoManager

To create an AutoManager, you will first need to create a queue using the `generate_queue()` method. Then, an object of *AutoManager* can be created using the *create()*. Since this manager can only solve one kind of recaptcha per instance, you will need to pass the captcha details during instantiation. Example for creating an *AutoManager* that solves a recaptcha v2 captcha:

```
from recaptcha_manager.api import AutoManager

request_queue = recaptcha_manager.api.generate_queue()
manager =  AutoManager.create(request_queue, url='https://full.domain.here', sitekey=
↪'xxxx',
                                          captcha_type='v2')
```

Example for creating an *AutoManager* for solving recaptcha v3 captcha:

```
from recaptcha_manager.api import AutoManager

request_queue = recaptcha_manager.api.generate_queue()
manager =  AutoManager.create(request_queue, url='https://full.domain.here', sitekey=
↪'xxxx',
                                                captcha_type='v3', action='recaptcha-
↪action', min_score=0.7)
```

4.1.1 Sending, and receiving captcha requests

To signal *AutoManager* to send more captcha requests, you can use the *send_request()* method. It analyses collected data about your program's captcha usage and sends the optimal number of captcha requests to the solving service in the background automatically. If the analysis determines that no new captcha requests need to be sent, then *send_request()* does not send any, even if it is repeatedly called. So you can (and should) call this method regularly without any risk for over-sending attached. However, keep in mind that incase there isn't enough data to analyse, *AutoManager* simply sends a pre-defined number of request(s) (default is one). This may result in higher waiting times when you request the answers to the first few captchas using a newly created AutoManager. If this bothers you, then you can override the number of requests to send in such cases using the *initial* parameter:

```
request_queue = recaptcha_manager.api.generate_queue()
manager =  AutoManager.create(request_queue, url='https://full.domain.here', sitekey=
↪'xxxx',
                                                captcha_type='v2)
manager.send_request(initial=4)  # Instead of default 1, four requests will be sent if␣
↪data inadequate to make predictions
```

Next, to get a captcha answer, use the *get_request()* method. By default, it blocks until a captcha answer is available. However, internal fail-safes make sure that there are adequate captcha requests being solved, sending more whenever necessary, to prevent an infinite block. Overtime, as *AutoManager* collects more data, this block time will become almost negligible. Lastly, if the manager is *stopped*, and all available captcha requests have been used, *get_request()* will raise a *Exhausted* exception, signalling that this instance of *AutoManager* is no longer usable. Example code to properly receive captcha:

```
try:
   captcha = manager.get_request()
except recaptcha_manager.api.exceptions.Exhausted:
   print('No more captcha requests left')
else:
   print(f"Token is {captcha['answer']}")
   print(f"It cost ${captcha['cost']}")
```

Also, *get_request()* supports `max_block` as a parameter. If `max_block` provided is a non-zero value, then the function waits at most `max_block` seconds for a captcha answer to be available (if there is none already), after which it raises *TimeOutError* and returns control back to you. Keep in mind, however, that `max_block` should be used with caution since it may skew the data collected by the manager. Therefore, it is advised to not use a value lower than 60 if you are using `max_block` parameter. Example of using `max_block`:

```
try:
   captcha = manager.get_request(max_block=60)
except recaptcha_manager.api.exceptions.Exhausted:
   print('No more captcha requests left')
except recaptcha_manager.api.exceptions.TimeOutError:
   print('Timed out!')
else:
   print(f"Token is {captcha['answer']}")
   print(f"It cost ${captcha['cost']}")
```

Note: As a best practice, you should always call *send_request()* everytime before you call *get_request()*

4.1.2 Stopping the AutoManager

When you no longer need new recaptcha tokens, you can call *stop()* after which no new captcha requests will be sent even if you call *send_request()*. However, requests already solved, or currently being solved by the captcha service, will not be affected. Once all requests have been solved, AND used, only then will the manager no longer be usable. All subsequent calls to *get_request()* will then raise *Exhausted* exception.

Alternatively, you can use *force_stop()* as well. Unlike the simple *stop()*, force stopping the manager means that all solved captcha requests, including those which are in the process of being solved, are immediately discarded. All subsequent calls to receive captcha answers via *get_request()* will then immediately raise *Exhausted*. Keep in mind that both these methods can only be called once per manager, and *stop()* cannot be called if *force_stop()*

was already called. However, you can call *force_stop()* even if *stop()* was called before. For example, this is correct and doable:

```
manager.stop()
manager.force_stop()
```

But this is incorrect and will result in error:

```
manager.force_stop()
manager.stop()  # RuntimeError: "Manager is no longer usable or has already been force␣
↪stopped"
```

4.1.3 Restoration points

AutoManagers start collecting statistics the moment they are created, and continue to do so till they are stopped. During this entire cycle, *AutoManager* regularly removes older statistics and performs quality checks so it can adapt to any change of pace of your program if it so happens. However, incase of extended periods where your program does not need AutoManager, you should create restore points to restore the statistics back to their more accurate state when you were actually using the AutoManager. Doing so is particularly useful to "pause" the manager during lengthy, unforeseen errors, like waiting for network connectivity if it is lost.

To create a restore point, use *create_restore_point()*::

```
manager.create_restore_point(overwrite=False)
```

Keep in mind that only 1 restore point can be created at a time. If you want to overwrite a previously created restore point, then pass parameter *overwrite* as *True*. If *overwrite* is *False* and you attempt to create another restore point when one already exists, `RestoreError` will be raised. To restore *AutoManager* to the previously created restore point, use *AutoManager.restore()*:

```
manager.restore()
```

Attempting to restore without creating a restore point will result in `RestoreError`

4.1.4 Available captchas

Certain methods can be used to get information on how many captchas are being solved, or already have been solved. To find the number of captchas solved and available, use *AutoManager.available()*:

```
print(f'{manager.available()} captchas are solved and ready to be used')
```

To find the number of captchas that are currently being solved, use *AutoManager.being_solved()*:

```
print(f'{manager.being_solved()} captchas are currently being solved')
```

Note: The method number returned by `.being_solved()` is unreliable if you call it after *stopping the manager*. Additionally captchas currently being solved is not a reliable indicator of how many captchas will actually end up being solved. This is because the *service process* may encounter a *service-specific error* and quit, in which case all registered tasks will be lost.

4.1.5 Statistics

AutoManager provides access to several of the statistics it collects:

- Method *AutoManager.get_waiting_time()* returns the average time your program has to wait to receive captchas when calling *AutoManager.get_request()*. *AutoManager* tries to reduce this value to a 0.

```
print(f"Captchas available after waiting for an average of {manager.get_waiting_
↪time()}s")
```

- Method *AutoManager.get_solving_time()* returns the average time the solving service take to register, and solve the captcha.

```
print(f"Service takes an average of {manager.get_solving_time()}s to solve one␣
↪captcha")
```

- Method *AutoManager.get_use_rate()* returns the average time your program takes between successive calls to *AutoManager.get_request()*. It represents how frequently your program needs captchas.

```
print(f"One captcha is requested every {manager.get_use_rate()}s from the manager")
```

- Methods *AutoManager.get_solved()* and *AutoManager.get_used()* returns the total number of captchas that have been solved, and the total number that have been used respectively

```
print(f"Out of {manager.get_solved()} captchas solved, you have used {manager.get_
↪used()}")
```

- Method *AutoManager.get_expired()* returns how many captchas that had been solved ended up expiring because they were not used timely. *AutoManager* tries to keep this number as low as possible

```
print(f"A total of {manager.get_expired()} captchas were expired")
```

4.2 ManualManager

To create a manager, you will first need to create a queue using the `generate_queue()` method. Then, an object of *ManualManager* can be created using the *create()*.

```
from recaptcha_manager.api import ManualManager

request_queue = recaptcha_manager.api.generate_queue()
manager =  ManualManager.create(request_queue)
```

4.2.1 Sending, and receiving captcha requests

To signal *ManualManager* to send more captcha requests, you can use the *send_request()* method, passing along the appropriate captcha parameters and the number of such captchas you wish to solve. The function would then return a string, referred to as the `batch_id` for the captcha(s) you just requested.:

```
# Example recaptcha v2
id = manager.send_request(url='https://my.target', sitekey='xxxx', captcha_type='v2',␣
↪number=2)
```

(continues on next page)

(continued from previous page)

```
# Example recaptcha v3
id = manager.send_request(url='https://my.target', sitekey='xxxx', captcha_type='v3',␣
↪action='home', min_score=0.7, number=2)
```

This `batch_id` is actually a hash of the parameters created with sha256, and will always be the same for all captchas requested with the same captcha parameters (the *number* parameter does not affect the `batch_id`), no matter when you call or where you call *send_request()* from. Note that if the provided url parameters have the same domain names, they are still considered same regardless of the full path. For example, consider 5 captchas, **Captcha A, B, C, D** and **E**, with the following captcha parameters:

name	type	url	sitekey
Captcha A	v2	https://domain.com	xxxxx
Captcha B	v2	https://domain.com/full/path	xxxxx
Captcha C	v2	https://domain.com/path	xxxxx
Captcha D	v3	https://domain.com	xxxxx
Captcha E	v2	https://differentsite.com	yyyyy

Out of these, **Captcha A, B** and **C** will generate identical `batch_id` while **Captcha D, E** will generate unique `batch_id` because of a unique combination of captcha-parameters when compared to others. If you want *ManualManager* to use the full url instead of just the domain when sending captcha request and generating `batch_id`, set parameter `force_path` as *True* when using *send_request()*. Doing so will force **Captcha A, B, C** to generate unique `batch_id` as well.

The `batch_id` generated can now be used to get answers for the particular captchas you want by providing them to *get_request()*. By default, this function blocks until a captcha answer is available, but will also raise *EmptyError* if no captcha task with the given `batch_id` is being solved and parameter `force_return` is set to *True* (the default value). Additionally, if the manager is stopped, and there are no longer any captcha answers left to be used, *get_request()* will raise an *Exhausted* exception, signalling that this instance of *ManualManager* is no longer usable:

```
try:
   captcha = manager.get_request(id=id)

except recaptcha_manager.api.exceptions.Exhausted:
   print('No more captcha requests left, the manager is no longer usable')

except recaptcha_manager.api.exceptions.Empty:
   print('No requests being solved for provided id. Try sending more requests')

else:
   print(f"Token is {captcha['answer']}")
   print(f"It cost ${captcha['cost']}")
```

Adding on, you can also specify the function a maximum time to wait, in seconds, for the captcha answer by using the `max_block` parameter. If no captcha is available in that time frame, *TimeOutError* exception will be raised. It is recommended that you at least set `max_block` to a non-zero value, or keep `force_return` as true to avoid a possibility for an infinite block. These two parameters can also be used together:

```
try:
   captcha = manager.get_request(id=id, max_block=30)

except recaptcha_manager.api.exceptions.Exhausted:
   print('No more captcha requests left')
```

(continues on next page)

(continued from previous page)

```
except recaptcha_manager.api.exceptions.TimeOutError:
   print('Maximum waiting time exceeded! Try sending more requests.')

except recaptcha_manager.api.exceptions.Empty:
   print('No requests being solved for provided id. Try sending more requests')

else:
   print(f"Token is {captcha['answer']}")
   print(f"It cost ${captcha['cost']}")
```

4.2.2 Stopping the ManualManager

When you no longer need new recaptcha tokens, you can call *stop()* after which no new captcha requests will be sent even if you call *send_request()*. However, requests already solved, or requests already sent and currently being solved by the captcha service, will not be affected. Once all requests have been solved, and then used as well, only then will the manager no longer be usable. All subsequent calls to *get_request()* after this will return a *Exhausted* error:

```
manager.stop()
```

Alternatively, you can use *force_stop()* as well. Unlike the simple *stop()*, force stopping the manager means that all already solved captcha requests, including those which are in the process of being solved, are immediately discarded. All subsequent calls to receive captcha answers via *get_request()* will then immediately raise *Exhausted*. Keep in mind that both these methods can only be called once per manager, and *stop()* cannot be called if *force_stop()* was already called. However, you can call *force_stop()* even if *stop()* was called before. For example, this is correct and doable:

```
manager.stop()
manager.force_stop()
```

But this is incorrect and will result in error:

```
manager.force_stop()
manager.stop()  # RuntimeError: "Manager is no longer usable or has already been force␣
↪stopped"
```

4.2.3 Available captchas

ManualManager provides a way to get to know the status of the captcha requests sent to the solving service for all *batch_ids*. To get the number of captchas that are currently being solved by the service for any `batch_id`, use *AutoManager.being_solved()*:

```
number = manager.being_solved(batch_id=id)
print(f'{number} captchas are currently being solved')
```

To get the number of captchas already solved and available, use *AutoManager.available()*:

```
number = manager.available(batch_id=id)
print(f'{number} captchas are solved and ready to be used')
```

Note: The method number returned by `.being_solved()` is unreliable if you call it after *stopping the manager*. Additionally captchas currently being solved is not a reliable indicator of how many captchas will actually end up being solved. This is because the *service process* may encounter a *service-specific error* and quit, in which case all registered tasks will be lost.

For both these methods, if you do not specify a `batch_id`, the manager will return the information requested for all `batch_id` instead.

CHAPTER

FIVE

SERVICE PROCESSES

Service processes are background processes used to communicate with the captcha solving service via their API. They are responsible for sending captcha requests to the solving services, and fetching the answers to them as well. Since they run in a different process, your program has limited control over them and most communication is done through the managers. This section goes into detail about how to correctly start and manage a service process, and all their available features.

5.1 Starting & stopping services

To start a service process you must first choose the solving service you want to use. Recaptcha-manager supports three: AntiCaptcha, 2Captcha and CapMonster. All three services have their own classes which behave identically. Additionally, you would also require a queue which can be created using *recaptcha_manager.api.generators.generate_queue()* function. However, if you have already created a manager, then use the same queue you passed during the creation of the manager. You can now create a service using the *create_service()* method by passing the queue and the solving service's API key, and then using the *BaseService.spawn_process()* to start the service process. A *very* basic example is given below:

```
from recaptcha_manager.api import AntiCaptcha, TwoCaptcha, CapMonster

# For Anticaptcha
service = AntiCaptcha.create_service(api_key='xxxxx', request_queue=queue)

# For Capmonster
service = CapMonster.create_service(api_key='xxxxx', request_queue=queue)

# For 2Captcha
service = TwoCaptcha.create_service(api_key='xxxxx', request_queue=queue)

service_proc = service.spawn_process()
```

That's it, the service process is now running in the background!

Note: Even though it's not disallowed, it is **not recommended** to spawn a service process without specifying an *exception handler*

Once you are done, you can stop the service process by using the *stop()* method. Keep in mind that the service process doesn't *immediately* stop upon calling the method. If you want to absolutely make sure that the service process is no longer running, you can wait for it to join by using *safe_join()*:

```
# Signal the service to stop
service.stop()

# Optionally, wait for the process to completely quit
# service.safe_join(service_proc)
```

By default, *safe_join()* waits as long as it takes for the service process to quit before returning, but you can set a timeout using `max_block` parameter. If you set it to a value greater than zero (0 is the default value and disables timeout), then the method attempts to join the service process for at most `max_block` seconds before returning. You can then check the service process's `exitcode` to determine whether it has finished or not:

```
# Signal the service to stop
service.stop()

# Attempt to join the service process for a maximum of 15s
service.safe_join(service_proc, max_block=15)

if service_proc.exitcode is None:
   print("Service process has not finished yet!")
else:
   print("Service process has finished with exitcode:", service_proc.exitcode)
```

5.2 Exception Handling

Like previously mentioned, service processes are expected to handle communication with the solving service API. This often involves connection errors and solving service specific errors that are likely to happen. Therefore, handling such errors is important to keep the service process running. Fortunately, recaptcha-manager provides a robust way to do so.

5.2.1 Service errors and outer-scope

Service specific errors include *LowBidError*, *NoBalanceError*, *BadAPIKeyError*, and *UnexpectedResponse*. These all are considered severe errors and are automatically raised to the outer scope. If an exception is raised to the outer scope, then the service process stops immediately until you restart it. Additionally, all captcha tasks registered with the captcha service will be lost as well. To get the exception which was raised to the outer scope, one can use *get_exception()* which re-raises the last such exception if it exists, otherwise returns `None` if no exception has been raised to the outer-scope since the last time the service process was run. Call this method periodically to make sure that the service process is running without issues. Example:

```
service = AntiCaptcha.create_service(api_key='xxxxx', request_queue=queue)
service_proc = service.spawn_process()

try:
   service.get_exception()

except recaptcha_manager.api.exceptions.LowBidError:
   print('Bid too low, raise it from your account settings!')

except recaptcha_manager.api.exceptions.NoBalanceError:
   print('Balance too low, refill from your account dashboard!')
```

(continues on next page)

(continued from previous page)

```
    raise

except recaptcha_manager.api.exceptions.BadAPIKeyError:
    print('API key provided is incorrect!')
    raise

else:
    print("Service process running smoothly!")
```

Keep in mind that recaptcha-manager is process-safe and uses shared memory (check *Share managers & services*). Therefore, you can check the service status from a different process with minimal changes to your main code if it suits you better. For example:

```
def service_checker(service):
    while True:
        try:
            service.get_exception()

        except recaptcha_manager.api.exceptions.LowBidError:
            print('Bid too low, raise it from account settings!')

        except recaptcha_manager.api.exceptions.NoBalanceError:
            print('Balance too low, refill before continuing!')

        except recaptcha_manager.api.exceptions.BadAPIKeyError:
            print('API key provided is incorrect!')

        time.sleep(10)  # Check status every 10 seconds


service = AntiCaptcha.create_service(api_key='xxxxx', request_queue=queue)
service_proc = service.spawn_process()

# We continuously check that the service process is running inside another process,␣
↪which does not disrupt our
# main process
checker = multiprocess.Process(target=service_checker, args=(service,))
checker.start()
```

If you wish to restart the service process once it is stopped, you can always do so using the same function:

```
service_proc = service.spawn_process()
```

Because service errors often require manual intervention (refilling of balance, increasing bid from account settings , etc.), resolving them is out of scope for recaptcha-manager. Best way to resolve these errors then is through prevention: make sure your service account balance is sufficient and the bid (if the service you use supports that) is adequate before running your program. Additionally, you can limit the effects service errors have by using *Multiple services* so that even incase one stops working, your program can still function.

5.2.2 Connection Errors

Connection errors like timeouts are common and may result in the service process stopping everytime they occur. Therefore, to handle connection errors, you can specify a callable which will be called everytime an exception occurs by using the *exc_handler* parameter when starting the service process with *spawn_process()*. The exception is then passed as an argument to this callable. Therefore, you can have your own code to handle the exceptions relating to connection errors.

Note: Recaptcha-manager uses `requests` under the hood to make the requests.

By default, after the exception is passed to `exc_handler`, it is assumed that the exception has been handled and the HTTP request that raised the exception will then be retried automatically. Therefore, the callable you pass as `exc_handler` must raise the exceptions that it cannot handle to the outer scope. This will stop the service process till you restart it. Sample handlers below demonstrate two different approaches to do this, where one raises all errors except a few, and the other ignores all errors except a few::

```
def exc_handler(exc):
    '''All errors except NonFatalConnectionError and SomeOtherNonFatalError will be raised!'
↪'

    if isinstance(exc, NonFatalConnectionError):
        pass  # Ignore this error, after which the service process will resend the␣
↪request

    elif isinstance(exc, SomeOtherNonFatalError):
        pass # We ignore this one too

    else:
        raise  # Remember, all other errors that we don't handle or don't know about, we␣
↪should raise!


def exc_handler_two(exc):
   '''All errors except FatalConnectionError and SomeOtherFatalError will be IGNORED! If␣
↪you decide to do this
      then make sure to atleast log them somewhere to aid in debugging'''

   if isinstance(exc, FatalConnectionError):
        raise  # raise this error since we can't handle it. This will stop the service␣
↪process till you restart it.

   elif isinstance(exc, SomeOtherFatalError):
        raise # We raise this one too

   else:
        log_error(exc)
        pass  # All other errors we ignore!
```

Similarly, if you want to automatically retry all requests that raised errors, you can ignore the exceptions raised by default as well

```
def exc_handler_three(exc):
   '''Ignore all errors and automatically retry the requests till they succeed. If you␣
↪decide to do this then make sure to atleast log them somewhere to aid in debugging'''
```

(continues on next page)

(continued from previous page)

```
    log_error(exc)
    return
```

Note: Incase no *exc_handler* is provided, then all exceptions will automatically be raised to the outer scope.

Additionally, you can pass a `Retry` object which will be mounted to every outgoing request (see parameter `retry` in *spawn_process()*):

```
from requests.packages.urllib3.util.retry import Retry

retries = Retry(total=5, backoff_factor=1)
```

You can then pass this to the service process:

```
service_proc = service.spawn_process(retry=retries, exc_handler=exc_handler)
```

5.3 Multiple services

You can use multiple services with recaptcha-manager simultaneously. Even further, you can also control which managers use which services if multiple of them are running. No extra configurations are required to use multiple services, you just simply start two services instead of one with the same queue and use them normally.:

```
queue = recaptcha_manager.api.generate_queue()

# Start the anticaptcha service
anticap = AntiCaptcha.create_service(api_key='xxxxx', request_queue=queue)
anticap_proc = service.spawn_process(exc_handler=exc_handler)

# Start the 2Captcha service
twocap = TwoCaptcha.create_service(api_key='yyyy', request_queue=queue)
twocap_proc = service.spawn_process(exc_handler=exc_handler)
```

Any managers now created using this queue would now send their requests to either anticaptcha or 2captcha, whichever gets the request first.

If there are multiple services running, and if you want to create managers that only send captcha requests to particular service(s), you can do that by creating multiple queues, and passing the same queue to the particular manager and the service during creation:

```
queue_anticap = recaptcha_manager.api.generate_queue()
queue_twocap = recaptcha_manager.api.generate_queue()

# Start the anticaptcha service with one of those queues
anticap = AntiCaptcha.create_service(api_key='xxxxx', request_queue=queue_anticap)
anticap_proc = service.spawn_process(exc_handler=exc_handler)

# Start the 2Captcha service with the other queue
twocap = TwoCaptcha.create_service(api_key='yyyy', request_queue=queue_two_cap)
twocap_proc = service.spawn_process(exc_handler=exc_handler)
```

(continues on next page)

(continued from previous page)

```
# This manager will always send any and all captcha requests to anticaptcha service,␣
↪because both of them share the # same queue
anticap_manager = AutoManager.create(anticap_queue, url='https://full.domain.here',␣
↪sitekey='xxxx',
                                     captcha_type='v2)

# And this will send to the TwoCaptcha service
twocap_manager = AutoManager.create(twocap_queue, url='https://full.domain.here',␣
↪sitekey='xxxx',
                                    captcha_type='v2)
```

CHAPTER

SIX

MULTIPROCESSING AND RECAPTCHA-MANAGER

Recaptcha-manager uses `multiprocess`, a fork of `multiprocessing` to ensure non-blocking code, and is designed keeping parallelism in mind. This section is aimed to inform you about how recaptcha-manager uses multiprocessing, best practices associated with it, and how you can customize it's use of multiprocessing according to your needs.

6.1 Share managers & services

Recaptcha-manager already uses shared memory and internal synchronization primitives to make managers and services process and thread safe. Unless specified, you can assume this is true for the entirety of recaptcha-manager's public API. Therefore, you can pass instances of managers and services to different processes and still be able to use them like you would normally do.

For example, instead of creating a separate manager in each process, you should create one and pass it to other processes. The data inside the manager will automatically be synchronized across all processes that access it. Moreover, the manager you have access to is a proxy object, so it will be quicker to pickle and pass to other processes as well. Sharing managers like this instead of creating one per process has the added benefit that the manager will have access to more consolidated data while using lesser resources. Example of sharing managers using a `multiprocessing.Pool`:

```
def worker(manager):
    # Do something with the manager
    return manager.available()

if __name__ == "__main__":
    pool = Pool(8)

    # Create a manager
    url = 'https://some.domain.com'
    sitekey = 'xxxxxxx'
    captcha_type = 'v2'
    request_queue = recaptcha_manager.api.generate_queue()
    manager = AutoManager(request_queue, url, sitekey, captcha_type)

    # Start 8 tasks which require managers. The manager will be synchronized across␣
↪processes automatically!
    for _ in range(8):
        results = pool.map(worker, [manager]*8)
```

6.2 Joining service processes

You should join the service process you created after stopping the service. This ensures proper cleanup and any resources used by the process will hence be properly released back. However, beware of joining the service process normally, since that may cause a dead lock if the service process raised an error to the outer-scope before terminating. Instead, you should use the *safe_join()* method whenever you want to join the service process. A minified example:

```
if __name__ == "__main__":
    request_queue = generate_queue()
    service = TwoCaptcha.create_service(API_KEY, request_queue)
    service_proc = service.spawn_process()

    service.stop()
    service.safe_join(service_proc)
```

6.3 Using standard library's multiprocessing

While `multiprocess` is a convenient fork of the built-in multiprocessing, these two libraries aren't fully compatible with each other. Trying to integrate recaptcha-manager in a project which uses the built-in multiprocessing rather than `multiprocess`, can then become difficult.

To support such use-cases, you can configure recaptcha-manager to use the built-in multiprocessing instead. Example:

```
from recaptcha_manager import configurations
# Setting to False means to use built-in multiprocessing. Default is True, which means
# to use multiprocess.
configurations.USE_DILL = False

# Now you can import .api sub-package, it will use the built-in multiprocessing instead
from recaptcha_manager.api import AutoManager, generate_queue
```

Keep in mind that you must edit the configurations before you import anything from within `recaptcha_manager.api`! Editing it after importing will have no effect.

6.4 Passing managers when generating queues

By default, whenever you generate a queue, a `multiprocessing.Manager` is spawned (not to be confused with the managers like `recaptcha_manager.api.managers.AutoManager` and `recaptcha_manager.api.managers.ManualManager` that recaptcha-manager offers). Therefore, if you are planning on using many queues and want to handle the resources yourself, then you may spawn a manager yourself and pass it when generating a queue. It will then use that manager to create the queue:

```
import multiprocess # or multiprocessing, if you have changed the configurations already
from recaptcha_manager.api import generate_queue

if __name__ == "__main__":
    multiprocess_manager = multiprocess.Manager()
    request_queue = generate_queue(manager=multiprocess_manager)
```

Keep in mind, however, that while creating multiple queues from the same manager will use lesser resources, it will adversely impact the performance of the queues. Lastly, make sure that the manager you create is from the correct package. Recaptcha-manager uses `multiprocess` by default, however, if you changed the configurations to use the standard library's py:mod:*multiprocessing* instead, then you must create the manager using `multiprocessing.Manager()` instead (note the -ing). If there is a discrepancy between the package recaptcha-manager is configured to use and the one you used to create the manager, then it is likely that an `multiprocessing.AuthenticationError` will be raised down the road when the queue is used.

CHAPTER

SEVEN

TESTING

From inside the project root, run:

```
python -m unittest
```

CHAPTER

EIGHT

BACKWARDS COMPATIBILITY

Recaptcha-manager's API is in active development, and is not yet stable. This means that new features are being added, some of which may break backwards compatibility. While changes to code that breaks backwards compatibility with previous versions are rare, they may happen to improve stability of the package in future. For convenience, an exhaustive list of such changes is provided below. Check this section regularly to stay updated on the latest changes so you can implement them as soon as possible.

8.1 Version 0.0.7 and above

Content of modules *generators.py*, *exceptions.py*, *manager.py*, and *services.py* were shifted to a *api* sub-package. What this results in is that importing directly from *recaptcha_manager* will no longer work, you would instead need to import from *recaptcha_manager.api*. Consider the below import statements that would work in previous versions:

```
from recaptcha_manager import AutoManager, TwoCaptcha, generate_queue
from recaptcha_manager.exceptions import LowBidError, Exhausted
```

To make them compatible with the newer versions, change them to this:

```
from recaptcha_manager.api import AutoManager, TwoCaptcha, generate_queue
from recaptcha_manager.api.exceptions import LowBidError, Exhausted
```

8.2 Version 0.0.3 - 0.0.6

Version 0.0.3 included a major update to Managers and services. These changes are documented separately for convenience

8.2.1 Changes in AutoManagers

- Method get_upcoming is no longer available. To get status on captcha requests, refer to section *Available captchas*.
- Upon *stopping* the manager, when there are no more requests left, *AutoManager.get_request()* will raise *Exhausted* instead of queue.Empty.
- If the captcha solving service reported an error with the captcha information you provided to the manager, then the error will be raised when you request the captcha using *AutoManager.get_request()* rather than in the service process.

8.2.2 Changes in Service Processes

- Flags are no longer needed to create service processes. Refer to *this* section for details on stopping service processes.
- Unlike previously, instances of the services needs to be created before you can *start a service process*. Consider this code below which would work in previous versions to start a service process:

```
flag = recaptcha_manager.generate_flag()
queue = recaptcha_manager.generate_queue()
key = 'xxxxxxx'

service_process = recaptcha_manager.AntiCaptcha.spawn_process(flag=flag,␣
↪request_queue=queue, APIKey=key, exc_handler=exc_handler)
```

  Equivalent of this code for version 0.0.3 and above:

```
queue = recaptcha_manager.generate_queue()
key = 'xxxxxxx'

service = recaptcha_manager.AntiCaptcha.create_service(request_queue=queue,␣
↪key=key)
service_process = service.spawn_process(exc_handler=exc_handler)
```

- Keyword argument `state`, which was passed when spawning a service process, is no longer supported. If a service process quits, all registered captcha tasks will be lost. This was done to localize service processes which would otherwise lead to unexpected bugs.
- Contrary to previous versions, if an `exc_handler` is passed, then the service process will ignore *Connection Errors* if they are not explicitly raised within the `exc_handler` callable. Previously, all connection errors would have been automatically raised unless you explicitly asked them not to by returning a Truthy value. For example, consider this code written for previous versions:

```
def exc_handler(exc):
   '''All errors except SomeNonFatalError will be raised!'''

   if isinstance(exc, SomeNonFatalError):
      print('This error will be ignored!')
      return True  # Because we return True, this error will not be raised!

   else:  # If its not SomeNonFatalError raise it in outer scope
      return False
```

  The equivalent of this `exc_handler` for versions 0.0.3 and above is:

```
def exc_handler(exc):
   '''All errors except SomeNonFatalError will be raised!'''

   if isinstance(exc, SomeNonFatalError):
       print('This error will be ignored!')
```

(continues on next page)

(continued from previous page)

```
    else:
        raise
```

- You no longer need to create your own wrapper to retrieve exceptions raised in the service process. Check this *section* for handling such exceptions.

CHAPTER

NINE

REFERENCES

This section contains all relevant code and its documentation separated by their classes

9.1 Low-level classes

class recaptcha_manager.api.services.**BaseService**(*key*, *request_queue*, *proxy_ini=False*)
Base class for all Services. Acts as an interface between your program and captcha service

classmethod create_service(**args*, ***kwargs*)
Properly initializes a class instance.

Returns Service object which can be used to start a service process

Return type *BaseService*

get_exception()
If an exception has been raised in the service process, this re-raises the exception in the process that calls this function. Otherwise, returns None

is_alive()
Check whether the service process is alive

Returns Whether the service process is still running or not

Return type bool

is_stopped()
Check whether the service has been stopped.

Returns Whether the service has been asked to stop or not

Return type bool

requests_manager(*exc_handler=None*, *retry=None*, *disable_insecure_warning=True*)
Main function responsible for reading requests from request_queue and sending tasks to appropriate solving services.

Parameters

- **exc_handler** (*callable*) – An optional user-defined function which runs whenever an exception occurs.
- **retry** (*requests.packages.urllib3.util.retry.Retry*) – Retry object to be added to each request
- **disable_insecure_warning** (*boolean*) – Whether to disable urllib3.exceptions.InsecureRequestWarning

Keep in mind that this function blocks until the service is stopped. Therefore, if you are calling this method directly, it must be started in a different process than the main program.

safe_join(*service_proc*, *max_block=0*, *return_exceptions=False*)
Properly attempts to join the service process within max_block seconds. If max_block is 0, then will wait until process joins before returning. If service process returned an exception, and return_exceptions is True, then will re-raise the exception. Will return the exitcode of the service process.

Returns service_proc.exitcode. Will be None if the process is not finished yet

Parameters

- **service_proc** (*multiprocessing.Process*) – The started service process.
- **max_block** – Maximum time to wait. Set as 0 to disable timeout. Default value is 0.
- **return_exceptions** (*boolean*) – Whether to raise any exceptions caught from the service process.

spawn_process(*retry=None*, *exc_handler=None*, *disable_insecure_warning=True*) → multiprocess.context.Process
Wrapper for starting the background service process.

Parameters

- **retry** (*urllib3.util.Retry*) – Retry object to be mounted to each request
- **exc_handler** (*callable*) – An optional user-defined function which runs whenever an exception occurs. Defaults to None
- **disable_insecure_warning** (*boolean*) – Whether to disable InsecureRequestWarning

Returns Started solving service process

Return type multiprocessing.Process

The optional exc_handler parameter takes a callable which is called everytime an exception occurs. The exception is passed as a parameter to the callable. By default, after the exception occurs and exc_handler has been called, the request that raised the exception is retried. However, you can raise the exception from within the handler in which case the service process will quit.

stop()
Stops the service

class recaptcha_manager.api.manager.**BaseRequest**(*request_queue*, *maximum=0*, *initial=1*, *limit=0*)
Base class for managers

available()

Get how many captchas are available to be used

Return type int

being_solved()

Get how many captchas are being currently solved

Return type int

classmethod create(**args*, ***kwargs*)
Properly initializes a class instance.

Returns A proxy instance of class. Has same functionality as a regular instance and can share state between processes.

Return type ObjProxy

`flush()`
Remove all stored solved captchas (if any). Good for cleaning up after you are done with the manager

`force_stop()`
Stops production of new captcha requests and immediately stops the manager. Requests already solved, or currently being solved, will be discarded. Any new captcha requests sent after this method call will be rejected.

`get_expired()`
Returns How many total captchas, which were solved, expired before being used

Return type int

`get_solved()`
Returns how many total captchas have been solved by the manager

Return type int

`get_used()`
Returns how many total captchas have been used by your program

Return type int

`stop()`
Stops production of new captcha requests. Requests already being solved won't be affected and captcha tokens for those requests will be produced normally. Should be called when you no longer intend to send new requests. Any new captcha requests sent after this method call will be rejected.

9.2 Service classes

`class recaptcha_manager.api.services.AntiCaptcha(key, request_queue, proxy_ini=False)`
Bases: *recaptcha_manager.api.services.BaseService*

Uses Anticaptcha captcha service to solve recaptchas Contains all methods of the base class | URL: https://anti-captcha.com | Documentation: https://anti-captcha.com/apidoc

`classmethod create_service(key, request_queue)`
Properly initializes a class instance.

Parameters

- **key** (*string*) – API key of the solving service
- **request_queue** – Queue for communication with managers

Return type *AntiCaptcha*

`class recaptcha_manager.api.services.TwoCaptcha(key, request_queue, proxy_ini=False)`
Bases: *recaptcha_manager.api.services.BaseService*

Uses 2Captcha captcha service to solve recaptchas Contains all methods of the base class

URL: https://2captcha.com
Documentation: https://2captcha.com/2captcha-api

classmethod create_service(*key*, *request_queue*)
Properly initializes a class instance.

Parameters

- **key** (*string*) – API key of the solving service
- **request_queue** – Queue for communication with managers

Return type *TwoCaptcha*

class recaptcha_manager.api.services.**CapMonster**(*key*, *request_queue*, *proxy_ini=False*)
Bases: *recaptcha_manager.api.services.AntiCaptcha*

Uses Capmonster service to solve captcha. Contains all methods of the base class

URL: https://capmonster.cloud

classmethod create_service(*key*, *request_queue*)
Properly initializes a class instance.

Parameters

- **key** (*string*) – API key of the solving service
- **request_queue** – Queue for communication with managers

Return type *CapMonster*

9.3 Managers

class recaptcha_manager.api.manager.**AutoManager**(*request_queue*, *url*, *web_key*, *captcha_type*, *action=None*, *min_score=None*, *invisible=False*, *initial=1*, *maximum=0*, *limit=0*)
Manages creation and sending of captcha requests to service process. Predicts the optimal number of captchas to send based on usage statistics.

Note: Do not call the constructor directly, managers should be created through *create()* function

Example instantiation:

```
url = 'https://some.domain.com'
sitekey = 'xxxxxxx'
captcha_type = 'v2'  # or 'v3'

if __name__ == '__main__':
    request_queue = recaptcha_manager.generate_queue()
    manager = AutoManager.create(request_queue, url, sitekey, captcha_type)
```

PROXY
alias of *recaptcha_manager.api.manager.AutoManager.PROXY*

available()

Get how many captchas are available to be used

Return type int

being_solved()

Get how many captchas are being currently solved

Return type int

classmethod create(*request_queue*, *url*, *web_key*, *captcha_type*, *action=None*, *min_score=None*, *invisible=False*, *initial=1*, *maximum=0*, *limit=0*)

Properly initializes the constructor for AutoManager.

Parameters

- **request_queue** (*multiprocessing.Queue*) – A queue for communication between service process and managers. Can be generated by `recaptcha_manager.generate_queue()`
- **url** (*str*) – URL of target website
- **web_key** (*str*) – sitekey of target website
- **captcha_type** (*str*) – Version of recaptcha the target site uses. Can be 'v2' or 'v3'
- **action** (*str*) – Action parameter in case solving recaptcha v3
- **min_score** (*float*) – minimum score you want if solving recaptcha v3. Should be between 0 and 1
- **invisible** (*bool*) – Whether the target site uses invisible recaptcha v2
- **initial** (*int*) – Number of captcha requests to send when calling *send_request()* initially when there isn't enough data. Defaults to 1
- **maximum** (*int*) – Maximum number of captcha requests to send on one call of *send_request()* function. Set as 0 to specify no such limit.
- **limit** (*int*) – Maximum number of allowed captcha requests being solved at once. Set as 0 to disable this limit

Returns A proxy instance of class AutoManager. Has same functionality as a regular manager

Return type *AutoManager*

The number of captchas requests to send are predicted on the basis of usage details only if sufficient number (3) of captchas have been solved and used. Until then, `initial` (default value 1) number of captchas will be sent on every call of *send_request()* function.

create_restore_point(*overwrite=False*)

Create a copy of current statistics which can be used to restore the manager's state at a later point

Parameters **overwrite** – Whether to overwrite any existing restore points

flush()

Remove all stored solved captchas (if any). Good for cleaning up after you are done with the manager

force_stop()

Stops production of new captcha requests and immediately stops the manager. Requests already solved, or currently being solved, will be discarded. Any new captcha requests sent after this method call will be rejected.

get_expired()

Returns How many total captchas, which were solved, expired before being used

Return type int

get_request(*send_custom_reqs=True*, *max_block=0*)
Returns a solved captcha. Blocks until one is ready

Parameters

- **send_custom_reqs** (*bool*) – By default function will send additional captcha requests if there are none being solved. Set as False to prevent this
- **max_block** (*int*) – Maximum time the function blocks in seconds. Set as 0 to block until a request is recieved

Returns A dictionary containing the token under key 'answer'

Return type dict

Example

```
try:
    c = manager.get_request()  # Blocks until one is ready
except recaptcha_manager.api.exceptions.Exhausted:
    print('no more requests available')
else:
    token = c['answer']
```

Note: If `send_custom_reqs` is `False`, then code may block indefinitely if there aren't any captcha requests being solved. In case it is set to `False`, set `max_block` to a non-zero value

get_solved()
Returns how many total captchas have been solved by the manager

Return type int

get_solving_time()
Returns recent average time taken by the solving service to solve a captcha. Will be zero if not enough statistics collected.

Return type float

get_use_rate()
Returns how frequently your program requires recaptcha tokens (in seconds). Will be zero if not enough statistics collected.

Return type float

get_used()
Returns how many total captchas have been used by your program

Return type int

get_waiting_time()
Returns recent average waiting time to receive a captcha token from server process. Will be zero if not enough statistics collected.

Return type float

restore()
Revert the manager's statistics back using a created restore point

send_request(*maximum=None*, *initial=None*)
Predict and send optimal number of captcha requests to server process to minimize waiting time.

Parameters

- **maximum** (*int*) – Maximum number of requests to send. Overrides value passed when creating manager
- **initial** (*int*) – Number of requests to send if there is not enough data to predict. Overrides value passed when creating manager

This function must be called periodically to ensure the least waiting time. A general rule is to call it every time before you call *get_request()* function. If there are already enough requests sent, then the function will not send more to avoid captchas being expired.

stop()
Stops production of new captcha requests. Requests already being solved won't be affected and captcha tokens for those requests will be produced normally. Should be called when you no longer intend to send new requests. Any new captcha requests sent after this method call will be rejected.

class recaptcha_manager.api.manager.**ManualManager**(*request_queue*)

PROXY
alias of *recaptcha_manager.api.manager.ManualManager.PROXY*

available(*batch_id=None*)
Returns the number of captcha requests solved and available for use. If batch_id is provided, returns information for that particular batch_id only.

Parameters **batch_id** (*str*) – Optional parameter to restrict the lookup to a particular batch_id

being_solved(*batch_id=None*)
Returns the number of captcha requests being solved. If batch_id is provided, returns information for that particular batch_id only.

Parameters **batch_id** (*str*) – Optional parameter to restrict the lookup to a particular batch_id

classmethod create(*request_queue*)
Properly initializes instance.

Returns A proxy instance of class. Has same functionality as a regular instance and can share state between processes.

Return type *ManualManager*

flush()
Remove all stored solved captchas (if any). Good for cleaning up after you are done with the manager

force_stop()
Stops production of new captcha requests and immediately stops the manager. Requests already solved, or currently being solved, will be discarded. Any new captcha requests sent after this method call will be rejected.

get_expired()
Returns How many total captchas, which were solved, expired before being used

Return type int

get_request(*batch_id*, *max_block=0*, *force_return=True*)
Returns a solved captcha for the provided id. Blocks until one is ready or another condition reached.

Parameters

- **batch_id** (*str*) – The id of the type of captcha tasks you wish to retrieve
- **max_block** (*int*) – Maximum time the function blocks in seconds. Set as 0 to block until a request is received

- **force_return** (*bool*) – Whether to return None as soon as the number of captcha tasks being solved for the provided batch_id becomes zero. Takes precedence over max_block.

Returns A dictionary containing the token under key 'answer'

Return type dict

Example

```
try:
    c = manager.get_request(batch_id)  # Blocks until one is ready
except recaptcha_manager.api.exceptions.Exhausted:
    print('no more requests available')
else:
    token = c['answer']
```

Note: Make sure to either keep parameter *force_return* as True (default), or *max_block* as a non-zero value (or both) to avoid a possibility for an indefinite block time

get_solved()

Returns how many total captchas have been solved by the manager

Return type int

get_used()

Returns how many total captchas have been used by your program

Return type int

send_request(*url*, *web_key*, *captcha_type*, *number=1*, *action=None*, *min_score=None*, *invisible=False*, *force_path=False*)

Creates an id for the captcha requests and sends them to the service process. These requests will be solved by the captcha solving service in the background without interrupting your main program. Captcha requests with similar parameters will share the same id. Returns immediately.

Parameters

- **url** (*str*) – Full URL of the website where captcha is present.
- **web_key** (*str*) – Google sitekey of the captcha
- **captcha_type** (*str*) – Version of recaptcha. Can be 'v2' or 'v3' only.
- **number** (*int*) – Number of captcha requests to send for the specified parameters
- **action** (*str*) – The action string in case of solving recaptcha v3
- **min_score** (*float*) – The minimum recaptcha v3 score desired in case solving recaptcha v3. Should be between 0-1.
- **force_path** (*bool*) – Whether to take the entire URL (domain + path) in consideration when creating batch_id. If set to False, only uses domain.
- **invisible** (*bool*) – Whether the captcha is invisible recaptcha v2 or not.

Returns Returns the id of the created captcha requests.

Return type str

The returned id can then be used when calling the *get_request()* method to retrieve the answer for the captcha tasks created here, or for any other captcha tasks created with similar parameters in general.

stop()
Stops production of new captcha requests. Requests already being solved won't be affected and captcha tokens for those requests will be produced normally. Should be called when you no longer intend to send new requests. Any new captcha requests sent after this method call will be rejected.

9.4 Miscellaneous functions

recaptcha_manager.api.generators.**generate_queue**(*manager=None*)
Generates a proxy object of class Queue

Return type multiprocessing.Queue

9.5 Exceptions

exception recaptcha_manager.api.exceptions.**BadAPIKeyError**
Raised when server reports APIKey is incorrect

exception recaptcha_manager.api.exceptions.**BadDomainError**
Raised when server reports provided domain is incorrect. May also signify that provided sitekey-domain combination is incorrect

exception recaptcha_manager.api.exceptions.**BadSiteKeyError**
Raised when server reports provided sitekey is incorrect. May also signify that provided sitekey-domain combination is incorrect

exception recaptcha_manager.api.exceptions.**EmptyError**
Raised when no captchas are being currently solved for a specified batch_id when using ManualManagers

exception recaptcha_manager.api.exceptions.**Errors**
Base class for all recaptcha_manager exceptions

exception recaptcha_manager.api.exceptions.**Exhausted**
Raised when managers are no longer usable

exception recaptcha_manager.api.exceptions.**InvalidBatchID**
Raised when the batch_id supplied to ManalManager is incorrect

exception recaptcha_manager.api.exceptions.**LowBidError**
Only for captcha services which use a bidding system. Raised when client's bid is less than captcha-service's current required bid

exception recaptcha_manager.api.exceptions.**NoBalanceError**
Raised when server reports that the client's balance is insufficient

exception recaptcha_manager.api.exceptions.**RestoreError**
Raised due to an error when attempting to create or use restore points

exception recaptcha_manager.api.exceptions.**TimeOutError**
Raised when time spent waiting for a captcha inside managers exceeds the maximum allowed.

exception recaptcha_manager.api.exceptions.**UnexpectedResponse**
Raised when the solving service replied with an unparsable message

PYTHON MODULE INDEX

r

INDEX

I

L

M

N

P

R

S

T

U