
Repomate Documentation

Release 1.1.2

Simon Larsén

Mar 31, 2019

Contents:

1	Introduction	3
1.1	Philosophy and goals	3
1.2	Key concepts	3
1.3	Conventions	4
2	Install	5
2.1	Requirements	5
2.2	Check your Python version	5
2.3	Option 1: Install from PyPi with <i>pip</i>	5
2.4	Option 2: Clone the repo and the install with <i>pip</i>	6
3	repomate User Guide	7
3.1	Getting started (the <code>show-config</code> , <code>verify-settings</code> and <code>setup</code> commands)	7
3.2	Updating student repositories (the <code>update</code> command)	11
3.3	Opening and Closing issues (the <code>open-issues</code> and <code>close-issues</code> commands)	12
3.4	Cloning Repos in Bulk (the <code>clone</code> command)	14
3.5	Peer review (the <code>assign-reviews</code> and <code>purge-review-teams</code> commands)	14
3.6	Plugins for <code>repomate</code>	19
3.7	Migrate master repositories into the target (or master) organization (the <code>migrate</code> command)	21
4	Configuration	23
4.1	OAUTH token	23
4.2	Configuration file	23
5	CLI documentation	25
6	repomate Module Reference	27
6.1	<code>command</code>	27
6.2	<code>cli</code>	27
6.3	<code>config</code>	27
6.4	<code>exception</code>	28
6.5	<code>github_api</code>	29
6.6	<code>git</code>	29
6.7	<code>tuples</code>	30
6.8	<code>util</code>	31
6.9	Core plugins	33
6.10	Extension plugins	34

7 Indices and tables	35
Python Module Index	37

Important: Repomate is being renamed RepoBee. Please see <https://github.com/repobee/repobee> for details.

If you are new to repomate, the *Introduction* and *repomate User Guide* sections are must-reads. Developers looking to modify or utilize the core functionality in ways the CLI does not allow will be best served by looking at the modindex.

Important: If you use the *repomate User Guide* in any way and feel like skipping *Getting started (the show-config, verify-settings and setup commands)*, make sure to read *Configure Repomate for the target organization (show-config and verify-settings)* anyway! The rest of the guide assumes a configuration as described there.

Please open an issue and tag it with the `docs` tag for any bugs or missing information.

Repomate is an opinionated tool for managing anything from a handful to thousands of GitHub repositories for higher education courses. It was created as the old [teachers_pet](#) tool was getting long in the tooth, and the new [GitHub Classroom](#) wasn't quite what we wanted (we like our command line apps). Repomate is heavily inspired by [teachers_pet](#), but tries to both make for a more complete and streamlined experience.

1.1 Philosophy and goals

The primary goal of Repomate is to lower the bar for incorporating git and GitHub into higher education coursework, hopefully opening up the wonderful world of version control to teachers who may not be subject experts (and to their students). For new users, Repomate provides both a tool and an opinionated workflow to adopt. For the more experienced user, there is also opportunity to customize Repomate using its plugin system, which I am looking to expand even more. Repomate is primarily geared toward course administrators looking to generate repos for their students. Many features are however highly useful to teaching assistants, such as the ability to clone repos in bulk and perform arbitrary tasks on them (tasks can be implemented as plugins, see [Plugins for repomate](#)).

1.2 Key concepts

Some terms occur frequently in Repomate and are best defined up front. Some of the descriptions may not click entirely before reading the [repomate User Guide](#) section, so quickly browsing through these definitions and re-visiting them when needed is probably the best course of action.

- *Target organization*: The GitHub [Organization](#) related to the current course round.
- *Master repository*: Or *master repo*, is a template repository upon which student repositories are based.
- *Master organization*: The master organization is an optional organization to keep master repos in. The idea is to be able to have the master repos in this organization to avoid having to migrate them to the target organization for each course round. It is highly recommended to use a master organization if master repos are being worked on across course rounds.
- *Student repository*: Or *student repo*, refers to a *copy* of a master repo for some specific student.

- *GitHub instance*: A hosted GitHub service. This can be for example <https://github.com> or any Enterprise host.

1.3 Conventions

The following conventions are fundamental to working with Repomate.

- For each course and course round, use one target **Organization**.
- Any user of Repomate has unrestricted access to the target organization (i.e. is an owner).
- Master repos should be available as private repos in one of three places: - The master organization (recommended if the master repos are being maintained and improved across course rounds). - The target organization. If you are doing a trial run or have trivial (empty) master repos, this may be a good option. - Locally in the current working directory.
- Student repositories are copies of the default branches of the master repositories (i.e. `--single-branch` cloning is used by default). That is, until students make modifications.
- Student repositories are named `<username>-<master_repo_name>` to guarantee unique repo names.
- Each student is assigned to a team with the same name as the student's username. It is the team that is granted access to the repositories, not the student's actual user.
- Student teams have push access to the repositories, but not administrative access (i.e. students can't delete their own repos).

Note: Few of these conventions are actually enforced, and there are ways around almost every single one. However, with the exception of the *one organization per course round* convention, which must be ensured manually, Repomate will automatically adhere to the other conventions. Although Repomate does adhere to the conventions, there is no way to stop users from breaking them using e.g. the GitHub web interface, manually performing master repo migrations etc. Straying from the conventions may cause Repomate to behave unexpectedly.

2.1 Requirements

Repomate requires Python 3.5+ and a somewhat up-to-date version of git. Officially supported platforms are Ubuntu 17.04+ and macOS, but Repomate should run fine on any Linux distribution and also on [WSL](#) on Windows 10. Please report any issues with operating systems and/or git versions on the [issue tracker](#).

2.2 Check your Python version

For Repomate to run, you need to have Python 3.5 or later. On many operating systems, python is an alias for Python 2.7, and python3 is an alias for the latest version of Python 3 that is installed. For this install guide, python3 is assumed to be a Python version 3.5 or higher. You can check the version yourself with:

```
$ python3 --version
# or
$ python --version
```

2.3 Option 1: Install from PyPi with *pip*

The latest release of Repomate is on PyPi, and can thus be installed as usual with pip. I strongly discourage system-wide pip installs (e.g. `sudo pip install <package>`), as this may land you with incompatible packages in a very short amount of time. A per-user install can be done like this:

1. Execute `python3 -m pip install --user repomate` to install the package.
2. Run `repomate -h` to verify that you can find the script. - If that doesn't work, the `repomate` script can't be found. try `python3 -m repomate.main -h` to run the main module of Repomate (which is all the `repomate` script does anyway).

Important: A `--user` install will perform a local install for the current user. Any scripts will be installed in a user-local bin directory. If this directory is not on your path (which it often is not by default), you will not be able to run the `repomate` script (however, `python -m repomate.main` should still work). `pip` should issue a warning about this, including the path to the local bin directory. To resolve the problem, add the local bin directory to your `$PATH` variable.

2.4 Option 2: Clone the repo and the install with *pip*

If you want the dev version, you will need to clone the repo, as only release versions are uploaded to PyPi. Unless you are planning to work on this yourself, I suggest going with the release version.

1. **Clone the repo with git:**

- `git clone https://github.com/slarse/repomate`

2. **cd into the project root directory with `cd repomate`.**

3. **Install the requirements with `python3 -m pip install -r requirements.txt`**

- To be able to run the tests, you must install the `requirements.test.txt` file.

4. **Install locally with `pip`.**

- `python3 -m pip install --user .`, this will create a local install for the current user.
- Or just `pip install .` if you use `virtualenv`.
- For development, use `pip install -e .` in a `virtualenv`.

3.1 Getting started (the `show-config`, `verify-settings` and `setup` commands)

Important: This guide assumes that the user has access to a `bash` shell, or is tech-savvy enough to translate the instructions into some other shell environment.

The basic workflow of Repomate is best described by example. In this section, I will walk you through how to set up an [Organization](#) with master and student repositories by showing every single step I would perform myself. The basic workflow can be summarized in the following steps:

1. Create an organization (the target organization).
2. Configure Repomate for the target organization.
3. Verify settings.
4. Migrate master repositories into the target organization.
5. Create one copy of each master repo for each student.

There is more to Repomate, such as opening/closing issues, updating student repos and cloning repos in batches, but here we will just look at the bare minimum to get started. Now, let's delve into these steps in greater detail.

3.1.1 Create an organization

This is an absolutely necessary pre-requisite for using Repomate. Create an organization with an appropriate name on the GitHub instance you intend to use. You can find the `New organization` button by going to `Settings -> Organization`. I will call my *target organization* `repomate-demo`, so whenever you see that, substitute in the name of your target organization.

Important: At KTH, we most often do not want our students to be able to see each others' repos. By default, however, members have read access to *all* repos. To change this, go to the organization dashboard and find your way to Settings -> Member privileges. At the very bottom, there should be a section called Default repository permission. Set this to None to disallow students from viewing each others' repos unless explicitly given permission by an organization owner (e.g. you).

3.1.2 Configure Repomate for the target organization (show-config and verify-settings)

For the tool to work at all, it needs to be provided with an OAUTH2 token to whichever GitHub instance you intend to use. See the [GitHub OAUTH docs](#) for how to create a token. The token should have the `repo` and `admin:org` scopes. While we can set this token in an environment variable (see [Configuration](#)), it's more convenient to just put it in the configuration file, as we will put other default values in there. We can use the `show-config` command to figure out where to put the config file.

```
$ repomate show-config
[ERROR] FileNotFoundError: no config file found, expected location: /home/USERNAME/.config/
↳repomate/config.cnf
```

`show-config` will check that the configuration file exists and is syntactically correct. Well, technically it will try to load the config and fail to do so if it doesn't exist or is incorrectly formatted and then display it to the user. Here, the error message is telling use that it expected a config file at `/home/USERNAME/.config/repomate/config.cnf`, so let's add one there. It should look something like this:

```
[DEFAULTS]
github_base_url = https://some-enterprise-host/api/v3
user = slarse
org_name = repomate-demo
master_org_name = master-repos
token = SUPER_SECRET_TOKEN
```

Now, you need to substitute in some of your own values in place of mine.

- **Enter the correct url for your GitHub instance. There are two options:**
 - If you are working with an enterprise instance, simply replace `some-enterprise-host` with the appropriate hostname.
 - If you are working with `github.com`, replace the whole url with `https://api.github.com`.
- Replace `slarse` with your GitHub username.
- Replace `repomate-demo` with whatever you named your target organization.
- Replace `SUPER_SECRET_TOKEN` with your OAUTH token.
- Replace `master_org_name` with the name of the organization with your master repos. - If you keep the master repos in the target organization or locally, **remove this option**.

Important: The rest of this guide assumes the simplest possible setup of `_not_` having a separate master organization, but it is good practice to have the master repos separate for the sake of maintainability. If the master organization is configured in the config file, it won't matter for any but the `migrate` command (which you don't need then, anyway).

That's it for configuration, and we can check that the file is correctly found and parsed by running `show-config` again:

```
$ repomate show-config
[INFO] found valid config file at /home/slarse/.config/repomate/config.cnf
[INFO]
-----BEGIN CONFIG FILE-----
[DEFAULTS]
github_base_url = https://some-enterprise-host/api/v3
user = slarse
org_name = repomate-demo
master_org_name = master-repos
token = SUPER_SECRET_TOKEN
-----END CONFIG FILE-----
```

3.1.3 Verify settings

Now that everything is set up, it's time to verify all of the settings. Given that you have a configuration file that looks something like the one above, you can simply run the `verify-settings` command without any options.

```
$ repomate verify-settings
[INFO] verifying settings ...
[INFO] trying to fetch user information ...
[INFO] SUCCESS: found user slarse, user exists and base url looks okay
[INFO] verifying oauth scopes ...
[INFO] SUCCESS: oauth scopes look okay
[INFO] trying to fetch organization ...
[INFO] SUCCESS: found organization test-tools
[INFO] verifying that user slarse is an owner of organization repomate-demo
[INFO] SUCCESS: user slarse is an owner of organization repomate-demo
[INFO] trying to fetch organization master-repos ...
[INFO] SUCCESS: found organization master-repos
[INFO] verifying that user slarse is an owner of organization master-repos
[INFO] SUCCESS: user slarse is an owner of organization master-repos
[INFO] GREAT SUCCESS: All settings check out!
```

If any of the checks fail, you should be provided with a semi-helpful error message. When all checks pass and you get `GREAT SUCCESS`, move on to the next section!

3.1.4 Setting up master repos

How you do this will depend on where you want to have your master repos. I recommend having a separate, persistent organization so that you can work on repos across course rounds. If you already have a master organization with your master repos set up somewhere, and `master_org_name` is specified in the config, you're good to go. If you need to migrate repos into the target organization (i.e. you are not using a master organization), see the [Migrate master repositories into the target \(or master\) organization \(migrate command\)](#) section. For all commands but the `migrate` command, the way you set this up does not matter as Repomate commands go.

3.1.5 Setup student sepositories

Now that the master repos are set up, it's time to create the student repos. While student usernames *can* be specified on the command line, it's often convenient to have them written down in a file instead. Let's pretend I have three students with usernames `spam`, `ham` and `eggs`. I'll simply create a file called `students.txt` and type each username on a separate line.

```
spam
ham
eggs
```

An absolute file path to this file can be added to the config file with the `students_file` option (see [Configuration file](#)). Now, I want to create one student repo for each student per master repo. The repo names will be on the form `<username>-<master-repo-name>`, guaranteeing their uniqueness. Each student will also be added to a team (which bears the same name as the student's user), and it is the team that is allowed access to the student's repos, and not the student's actual user. That all sounded fairly complex, but again, it's as simple as issuing a single command with Repomate.

```
$ repomate setup -mn master-repo-1 master-repo-2 -sf students.txt
[INFO] cloning into master repos ...
[INFO] cloning into file:///home/slarse/tmp/master-repo-1
[INFO] cloning into file:///home/slarse/tmp/master-repo-2
[INFO] created team eggs
[INFO] created team ham
[INFO] created team spam
[INFO] adding members eggs to team eggs
[WARNING] user eggs does not exist
[INFO] adding members ham to team ham
[INFO] adding members spam to team spam
[INFO] creating student repos ...
[INFO] created repomate-demo/eggs-master-repo-1
[INFO] created repomate-demo/ham-master-repo-1
[INFO] created repomate-demo/spam-master-repo-1
[INFO] created repomate-demo/eggs-master-repo-2
[INFO] created repomate-demo/ham-master-repo-2
[INFO] created repomate-demo/spam-master-repo-2
[INFO] pushing files to student repos ...
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/ham-master-repo-2_
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/ham-master-repo-1_
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/spam-master-repo-1_
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/eggs-master-repo-2_
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/eggs-master-repo-1_
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/spam-master-repo-2_
↪master
```

Note that there was a `[WARNING]` message for the username `eggs`: the user does not exist. At KTH, this is common, as many (sometimes most) first-time students will not have created their GitHub accounts until sometime after the course starts. These students will still have their repos created, but the users need to be added to their teams at a later time (to do this, simply run the `setup` command again for these students, once they have created accounts). This is one reason why we use teams for access privileges: it's easy to set everything up even when the students have yet to create their accounts (given that their usernames are pre-determined).

And that's it, the organization is primed and the students should have access to their repositories!

3.2 Updating student repositories (the `update` command)

Sometimes, we find ourselves in situations where it is necessary to push updates to student repositories after they have been published. As long as students have not started working on their repos, this is fairly simple: just push the new files to all of the related student repos. However, if students have started working on their repos, then we have a problem. Let's start out with the easy case where no students have worked on their repos.

3.2.1 Scenario 1: Repos are unchanged

Let's say that we've updated `master-repo-1`, and that users `spam`, `ham` and `eggs` should get the updates. Then, we simply run `update` like this:

```
$ repomate update -mn master-repo-1 -s spam eggs ham
[INFO] cloning into master repos ...
[INFO] cloning into https://some-enterprise-host/repomate-demo/master-repo-1
[INFO] pushing files to student repos ...
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/spam-master-repo-1
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/eggs-master-repo-1
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/ham-master-repo-1
↪master
[INFO] done!
```

That's all there is to it for this super simple case. But what if `ham` had started working on `ham-master-repo-1`?

Note: Here, `-s spam eggs ham` was used to directly specify student usernames on the command line, instead of pointing to a student's file with `-sf students.txt`. All commands that require you to specify student usernames can be used with either the `-s|--students` or the `-sf|--students-file` options.

3.2.2 Scenario 2: At least 1 repo altered

Let's assume now that `ham` has started working on the repo. Since we do not force pushes (that would be irresponsible!) to the student repos, the push to `ham-master-repo-1` will be rejected. This is good, we don't want to overwrite a student's progress because we messed up with the original repository. There are a number of things one *could* do in this situation, but in Repomate, we opted for a very simple solution: open an issue in the student's repo that explains the situation.

Important: If we don't specify an issue to `repomate update`, rejected pushes will simply be ignored.

So, let's first create that issue. It should be a Markdown-formatted file, and the **first line in the file will be used as the title**. Here's an example file called `issue.md`.

```
This is a nice title

### Sorry, we messed up!
There are some grave issues with your repo, and since you've pushed to the
repo, you need to apply these patches yourself.

<EXPLAIN CHANGES>
```

Something like that. If the students have used `git` for a while, it may be enough to include the output from `git diff`, but for less experienced students, plain text is more helpful. Now it's just a matter of using `repomate update` and including `issue.md` with the `-i|--issue` argument.

```
$ repomate update -mn master-repo-1 -s spam eggs ham -i issue.md
[INFO] cloning into master repos ...
[INFO] cloning into https://some-enterprise-host/repomate-demo/master-repo-1
[INFO] pushing files to student repos ...
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/spam-master-repo-1
↪master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/eggs-master-repo-1
↪master
[ERROR] Failed to push to https://some-enterprise-host/repomate-demo/ham-master-repo-1
return code: 128
fatal: repository 'https://some-enterprise-host/repomate-demo/ham-master-repo-1/' not
↪found
[WARNING] 1 pushes failed ...
[INFO] pushing, attempt 2/3
[ERROR] Failed to push to https://some-enterprise-host/repomate-demo/ham-master-repo-1
return code: 128
fatal: repository 'https://some-enterprise-host/repomate-demo/ham-master-repo-1/' not
↪found
[WARNING] 1 pushes failed ...
[INFO] pushing, attempt 3/3
[ERROR] Failed to push to https://some-enterprise-host/repomate-demo/ham-master-repo-1
return code: 128
fatal: repository 'https://some-enterprise-host/repomate-demo/ham-master-repo-1/' not
↪found
[WARNING] 1 pushes failed ...
[INFO] Opening issue in repos to which push failed
[INFO] Opened issue ham-master-repo-1/#1-'Nice title'
[INFO] done!
```

Note that Repomate tries to push 3 times before finally giving up and opening an issue. This is because pushes can fail for other reasons than rejections, such as timeouts and other network errors.

Note: If you forget to specify the `-i|--issue` argument and get a rejection, you may simply rerun `update` and add it. All updated repos will simply be listed as `up-to-date`, and the rejecting repos will still reject the push! However, be careful not to run `update` with `-i` multiple times, as it will then open the same issue multiple times.

3.3 Opening and Closing issues (the `open-issues` and `close-issues` commands)

Sometimes, the best way to handle an error in a repo is to simply notify affected students about it. This is especially true if the due date for the assignment is rapidly approaching, and most students have already started modifying their repositories. Therefore, Repomate provides the `open-issues` command, which can open issues in bulk. When the time is right (perhaps after the deadline has passed), issues can be closed with the `close-issues` command.

3.3.1 Opening Issues

The `open-issues` command is very simple. Before we use it, however, we need to write a Markdown-formatted issue. Just like with the `update` command, the **first line of the file is the title**. Here is `issue.md`:

```
An important announcement

### Dear students
I have this important announcement to make.

Regards,
_The Announcer_
```

Awesome, that's an excellent issue. Let's open it in the `master-repo-2` repo for our dear students spam, eggs and ham, who are listed in the `students.txt` file (see *Setup student sepositories*).

```
$ repomate open-issues -mn master-repo-2 -sf students.txt -i issue.md
[INFO] Opened issue spam-master-repo-2/#1-'An important announcement'
[INFO] Opened issue eggs-master-repo-2/#1-'An important announcement'
[INFO] Opened issue ham-master-repo-2/#1-'An important announcement'
```

From the output, we can read that in each of the repos, an issue with the title `An important announcement` was opened as issue nr 1 (#1). The number isn't that important, it's mostly good to note that the title was fetched correctly. And that's it! Neat, right?

3.3.2 Closing Issues

Now that the deadline has passed for `master-repo-2`, we want to close the issues opened in *open*. The `close-issues` command takes a *regex* that runs against titles. All issues with matching titles are closed. While you *can* make this really difficult, closing all issues with the title `An important announcement` is simple: we provide the regex `\AAn important announcement\Z`.

```
$ repomate close-issues -mn master-repo-2 -sf students.txt -r '\AAn important_
↪announcement\Z'
[INFO] closed issue spam-master-repo-2/#1-'An important announcement'
[INFO] closed issue eggs-master-repo-2/#1-'An important announcement'
[INFO] closed issue ham-master-repo-2/#1-'An important announcement'
```

And there we go, easy as pie!

Note: Enclosing a regex expression in `\A` and `\Z` means that it must match from the start of the string to the end of the string. So, the regex used here *will* match the title `An important announcement`, but it will *not* match e.g. `An important announcement and lunch` or `Hey An important announcement`. In other words, it matches exactly the title `An important announcement`, and nothing else. Not even an extra space or linebreak is allowed.

3.3.3 Listing Issues

It can often be interesting to check what issues exist in a set of repos, especially so if you're a teaching assistant who just doesn't want to leave your trusty terminal. This is where the `list-issues` command comes into play. Typically, we are only interested in open issues, and can then use `list issues` like so:

```
$ repomate list-issues -mn master-repo-2 -sf students.txt
[INFO] spam-master-repo-2/#1: Grading Criteria created 2018-09-12 18:20:56 by glassey
[INFO] eggs-master-repo-2/#1: Grading Criteria created 2018-09-12 18:20:56 by glassey
[INFO] ham-master-repo-2/#1: Grading Criteria created 2018-09-12 18:20:56 by glassey
```

So, just grading criteria issues posted by the user `glassey`. What happened to the important announcements? Well, they are closed. If we want to see closed issues, we must specifically say so with the `--closed` argument.

```
$ repomate list-issues -mn master-repo-2 -sf students.txt --closed
[INFO] spam-master-repo-2/#2: An important announcement created 2018-09-17 17:46:43
↳by slarse
[INFO] eggs-master-repo-2/#2: An important announcement created 2018-09-17 17:46:43
↳by slarse
[INFO] ham-master-repo-2/#2: An important announcement created 2018-09-17 17:46:43
↳by slarse
```

Other interesting arguments include `--all` for both open and closed issues, `--show-body` for showing the body of each issue, and `--author <username>` for filtering by author. There's not much more to it, see `repomate list-issues -h` for complete and up-to-date information on usage!

3.4 Cloning Repos in Bulk (the `clone` command)

It can at times be beneficial to be able to clone a bunch of student repos at the same time. It could for example be prudent to do this slightly after a deadline, as timestamps in a `git` commit can easily be altered (and are therefore not particularly trustworthy). Whatever your reason may be, it's very simple using the `clone` command. Again, assume that we have the `students.txt` file from *Setup student repositories*, and that we want to clone all student repos based on `master-repo-1` and `master-repo-2`.

```
$ repomate clone -mn master-repo-1 master-repo-2 -sf students.txt
[INFO] cloning into student repos ...
[INFO] Cloned into https://some-enterprise-host/repomate-demo/spam-master-repo-1
[INFO] Cloned into https://some-enterprise-host/repomate-demo/ham-master-repo-1
[INFO] Cloned into https://some-enterprise-host/repomate-demo/ham-master-repo-2
[INFO] Cloned into https://some-enterprise-host/repomate-demo/eggs-master-repo-1
[INFO] Cloned into https://some-enterprise-host/repomate-demo/spam-master-repo-2
[INFO] Cloned into https://some-enterprise-host/repomate-demo/eggs-master-repo-2
```

Splendid! That's really all there is to the basic functionality, the repos should now be in your current working directory. There is also a possibility to run automated tasks on cloned repos, such as running test suites or linters. If you're not satisfied with the tasks on offer, you can define your own. Read more about it in the *Plugins for repomate* section.

3.5 Peer review (`assign-reviews` and `purge-review-teams` commands)

Peer reviewing is an important part of a programming curriculum, so of course Repomate facilitates this! The relevant commands are `assign-reviews` and `purge-review-teams`. Like much of the other functionality in Repomate, the peer review functionality is built around teams and limited access privileges. In short, every student repo up for review gets an associated peer review team generated, which has `pull` access to the repo. Each student then gets added to $0 < N < \text{num_students}$ peer review teams, and are to open a peer review issue in the associated repos. This is at least the the default. See *Selecting peer review allocation algorithm* for other available review allocation schemes.

Important: The commands `assign-peer-reviews`, `purge-peer-review-teams` and `check-peer-review-progress` have been renamed `assign-reviews`, `purge-review-teams` and `check-reviews`, respectively. The functionality is unchanged, and the old commands will continue to work until v2.0.0 is released. At that point, the old commands will be removed.

3.5.1 Getting started with peer reviews using `assign-reviews`

The bulk of the work is performed by `assign-reviews`. Let's have a look at the help message (i.e. run `repomate assign-reviews -h`):

```
$ repomate assign-reviews -h
usage: repomate assign-reviews [-h]
                                (-sf STUDENTS_FILE | -s STUDENTS [STUDENTS ...])
                                [-o ORG_NAME] [-g GITHUB_BASE_URL] [-t TOKEN]
                                [-tb] -mn MASTER_REPO_NAMES
                                [MASTER_REPO_NAMES ...] [-n N] [-i ISSUE]

For each student repo, create a review team with pull access named
<student>-<master_repo_name>-review and randomly assign other students to it.
All students are assigned to the same amount of review teams, as specified by
`--num-reviews`. Note that `--num-reviews` must be strictly less than the
amount of students.

optional arguments:
  -h, --help                show this help message and exit
  -sf STUDENTS_FILE, --students-file STUDENTS_FILE
                           Path to a list of student usernames.
  -s STUDENTS [STUDENTS ...], --students STUDENTS [STUDENTS ...]
                           One or more whitespace separated student usernames.
  -o ORG_NAME, --org-name ORG_NAME
                           Name of the target organization
  -g GITHUB_BASE_URL, --github-base-url GITHUB_BASE_URL
                           Base url to a GitHub v3 API. For enterprise, this is
                           usually `https://<HOST>/api/v3`
  -t TOKEN, --token TOKEN
                           OAUTH token for the GitHub instance. Can also be
                           specified in the `REPOMATE_OAUTH` environment
                           variable.
  -tb, --traceback          Show the full traceback of critical exceptions.
  -mn MASTER_REPO_NAMES [MASTER_REPO_NAMES ...], --master-repo-names MASTER_REPO_
  → NAMES [MASTER_REPO_NAMES ...]
                           One or more names of master repositories. Names must
                           either refer to local directories, or to master
                           repositories in the target organization.
  -n N, --num-reviews N
                           Assign each student to review n repos (consequently,
                           each repo is reviewed by n students). n must be
                           strictly smaller than the amount of students.
  -i ISSUE, --issue ISSUE
                           Path to an issue to open in student repos. If
                           specified, this issue will be opened in each student
                           repo, and the body will be prepended with user
                           mentions of all students assigned to review the repo.
                           NOTE: The first line is assumed to be the title.
```

Most of this, we've seen before. The only non-standard arguments are `--issue` and `--num-reviews`, the former of which we've actually already seen in the `open-issues` command (see [Opening Issues](#)). I will assume that both `--github-base-url` and `--org-name` are already configured in the configuration file (if you don't know what this mean, have a look at [Configuration file](#)). Thus, the only things we must specify are `--students/` `--students-file` and `--num-reviews` (`--issue` is optional, more on that later). Let's make a minimal call with the `assign-reviews` command, and then inspect the log output to figure out what happened. Recall that `students.txt` lists our three favorite students spam, ham and eggs (see [Setup student sepositories](#)).

```
$ repomate assign-reviews -mn master-repo-1 -sf students.txt --num-reviews 2
# step 1
[INFO] created team spam-master-repo-1-review
[INFO] created team eggs-master-repo-1-review
[INFO] created team ham-master-repo-1-review
# step 2
[INFO] adding members eggs, ham to team spam-master-repo-1-review
[INFO] adding members ham, spam to team eggs-master-repo-1-review
[INFO] adding members spam, eggs to team ham-master-repo-1-review
# steps 3 and 4, interleaved
[INFO] opened issue eggs-master-repo-1/#1-'Peer review'
[INFO] adding team eggs-master-repo-1-review to repo eggs-master-repo-1 with 'pull'
↪permission
[INFO] opened issue ham-master-repo-1/#2-'Peer review'
[INFO] adding team ham-master-repo-1-review to repo ham-master-repo-1 with 'pull'
↪permission
[INFO] opened issue spam-master-repo-1/#2-'Peer review'
[INFO] adding team spam-master-repo-1-review to repo spam-master-repo-1 with 'pull'
↪permission
```

The following steps were performed:

1. One review team per repo was created (`<student>-master-repo-1-review`).
2. Two students were added to each review team. Note that these allocations are `_random_`. For obvious reasons, there can be at most `num_students-1` peer reviews per repo. So, in this case, we are at the maximum.
3. An issue was opened in each repo with the title `Peer review`, and a body saying something like `You should peer review this repo..` The review team students were assigned to the issue as well (although this is not apparent from the logging).
4. The review teams were added to their corresponding repos with `pull` permission. This permission allows members of the team to view the repo and open issues, but they can't push to (and therefore can't modify) the repo.

That's it for the basic functionality. The intent is that students should open an issue in every repo they are to peer review, with a specific title. The title can then be regexed in the upcoming `check-review-progress` to see which students assigned to the different peer review teams have created their review issue. Of course, other schemes can be cooked up, but that is my current vision of how I myself will use it. Now, let's talk a bit about that `--issue` argument.

Important: Assigning peer reviews gives the reviewers read-access to the repos they are to review. This means that if you use issues to communicate grades/feedback to your students, the reviewers will also see this feedback! It is therefore important to remove the peer review teams (see [Cleaning with purge-review-teams](#)).

Specifying a custom issue

The default issue is really meant to be replaced with something more specific to the course and assignment. For example, say that there were five tasks in the `master-repo-2` repo, and the students should review tasks 2 and 3 based on some criteria. It would then be beneficial to specify this in the peer review issue, so we'll write up our own little issue to replace the default one. Remember that the first line is taken to be the title, in exactly the same way as issue files are treated in *Opening Issues*.

```
Review of master-repo-2

Hello! The students assigned to this issue have been tasked to review this
repo. Each of you should open _one_ issue with the title 'Peer review' and
the following content:

## Task 2
### Code style
Comments on code style, such as readability and general formatting.

### Time complexity
Is the algorithm  $O(n)$ ? If not, try to figure out what time complexity it is
and point out what could have been done better.

## Task 3
### Code style
Comments on code style, such as readability and general formatting.
```

Assuming the file was saved as `issue.md`, we can now run the command specifying the issue like this:

```
$ repomate assign-reviews -mn master-repo-2 -sf students.txt --num-reviews 2 --issue_
↪issue.md
[INFO] created team spam-master-repo-2-review
[INFO] created team eggs-master-repo-2-review
[INFO] created team ham-master-repo-2-review
[INFO] adding members ham, eggs to team spam-master-repo-2-review
[INFO] adding members spam, ham to team eggs-master-repo-2-review
[INFO] adding members eggs, spam to team ham-master-repo-2-review
[INFO] opened issue eggs-master-repo-2/#2-'Review of master-repo-2'
[INFO] adding team eggs-master-repo-2-review to repo eggs-master-repo-2 with 'pull'_
↪permission
[INFO] opened issue ham-master-repo-2/#2-'Review of master-repo-2'
[INFO] adding team ham-master-repo-2-review to repo ham-master-repo-2 with 'pull'_
↪permission
[INFO] opened issue spam-master-repo-2/#2-'Review of master-repo-2'
[INFO] adding team spam-master-repo-2-review to repo spam-master-repo-2 with 'pull'_
↪permission
```

As you can tell from the last few lines, the title is the one specified in the issue, and not the default title as it was before. And that's pretty much it for setting up the peer review repos.

3.5.2 Cleaning with `purge-review-teams`

The one downside of using teams for access privileges is that we bloat the organization with a ton of teams. Once the deadline has passed and all peer reviews are done, there is little reason to keep them (in my mind). Therefore, the `purge-review-teams` command can be used to remove all peer review teams for a given set of student repos. Let's say that we're completely done with the peer reviews of `master-repo-1`, and want to remove the review teams. It's as simple as:

```
$ repomate purge-review-teams -mn master-repo-1 -sf students.txt
[INFO] deleted team eggs-master-repo-1-review
[INFO] deleted team ham-master-repo-1-review
[INFO] deleted team spam-master-repo-1-review
```

And that's it, the review teams are gone. If you also want to close the related issues, you can simply use the `close-issues` command for that (see [Closing Issues](#)). `purge-review-teams` plays one more important role: if you mess something up when assigning the peer reviews. The next section details how you can deal with such a scenario.

3.5.3 Messing up and getting back on track

Let's say you messed something up with allocating the peer reviews. For example, if you left out a student, there is no easy way to rectify the allocations such that that student is included. Let's say we did just that, and forgot to include the student `cabbage` in the reviews for `master-repo-2` back at [Getting started with peer reviews using `assign-reviews`](#). We then do the following:

1. Check if any reviews have already been posted. This can easily be performed with `repomate list-issues -mn master-repo-2 -sf students.txt -r '^Peer review$'` (assuming the naming conventions were followed!). Take appropriate action if you find any reviews already posted (appropriate being anything you see fit to alleviate the situation of affected students possibly being assigned new repos to review).
2. Purge the review teams with `repomate purge-review-teams -mn master-repo-2 -sf students.txt`
3. Close all review issues with `repomate close-issues -mn master-repo-2 -sf students.txt -r '^Review of master-repo-2$'`
4. Create a new `issue.md` file apologetically explaining that you messed up:

```
Review of master-repo-2 (for real this time!)

Sorry, I messed up with the allocations previously. Disregard the previous
allocations (repo access has been revoked anyway).
```

5. Assign peer reviews again, with the new issue, with `repomate assign-reviews -mn master-repo-2 -sf students.txt --num-reviews 2 --issue issue.md`

And that's it! Disaster averted.

3.5.4 Selecting peer review allocation algorithm

The default allocation algorithm is as described in [Peer review \(`assign-reviews` and `purge-review-teams` commands\)](#), and is suitable for when reviewers do not need to interact with the students whom they review. This is however not always the case, sometimes it is beneficial for reviewers to interact with reviewees (is that a word?), especially if the peer review is done in the classroom. Because of this, Repomate also provides a `_pairwise_` allocation scheme, which allocates reviews such that if student A reviews student B, then student B reviews student A (except for an `A->B->C->A` kind of deal in one group if there are an odd amount of students). This implemented as a plugin, so to run with this scheme, you add `-p pairwise` in front of the command.

```
$ repomate -p pairwise assign-reviews -mn master-repo-1 -sf students.txt
```

Note that the pairwise algorithm ignores the `--num-reviews` argument, and will issue a warning if this is set (to anything but 1, but you should just not specify it). For more details on plugins in repomate, [Plugins for repomate](#).

3.6 Plugins for repomate

Repomate defines a fairly simple but powerful plugin system that allows programmers to hook into certain execution points. To read more about the details of these hooks (and how to write your own plugins), see the [repomate-plugin docs](#). Currently, plugins can hook into the `clone` command to perform arbitrary tasks on the cloned repos (such as running test classes), and the `assign-reviews` command, to change the way reviews are assigned.

3.6.1 Using Existing Plugins

You can specify which plugins you want to use either by adding them to the configuration file, or by specifying them on the command line. Personally, I find it most convenient to specify plugins on the command line. To do this, we can use the `-p|--plug` option *before* any other options. The reason the plugins must go before any other options is that some plugins add command line arguments, and must therefore be parsed separately. As an example, we can activate the *builtins* `javac` and `pylint` like this:

```
$ repomate -p pylint -p javac clone -mn master-repo-1 -sf students.txt
```

This will clone the repos, and then run the plugins on the repos. We can also specify the default plugins we'd like to use in the configuration file by adding the `plugins` option under the `[DEFAULTS]` section. Here is an example of using the *builtins* `javac` and `pylint`.

```
[DEFAULTS]
plugins = javac, pylint
```

Like with all other configuration values, they are only used if no command line options are specified. If you have defaults specified, but want to run without any plugins, you can use the `--no-plugins`, which disables plugins.

Important: The order plugins are specified in is significant and implies the execution order of the plugins. This is useful for plugins that rely on the results of other plugins. This system for deciding execution order may be overhauled in the future, if anyone comes up with a better idea.

Some plugins can be further configured in the configuration file by adding new headers. See the documentation of the specific plugins

3.6.2 Built-in plugins for repomate `assign-reviews`

Repomate ships with two plugins for the `assign-reviews` command. The first of these is the *defaults* plugin, which provides the default allocation algorithm. As the name suggests, this plugin is loaded by default, without the user specifying anything. The second plugin is the *pairwise* plugin. This plugin will divide N students into $N/2$ groups of 2 students (and possibly one with 3 students, if N is odd), and have them peer review the other person in the group. The intention is to let students sit together and be able to ask questions regarding the repo they are peer reviewing. To use this allocation algorithm, simply specify the plugin with `-p pairwise` to override the default algorithm. Note that this plugin ignores the `--num-reviews` argument.

3.6.3 Built-in Plugins for repomate `clone`

Repomate currently ships with two built-in plugins: `javac` and `pylint`. The former attempts to compile all `.java` files in each cloned repo, while the latter runs `pylint` on every `.py` file in each cloned repo. These plugins are mostly meant to serve as demonstrations of how to implement simple plugins in the `repomate` package itself.

pylint

The `pylint` plugin is fairly simple: it finds all `.py` files in the repo, and runs `pylint` on them individually. For each file `somefile.py`, it stores the output in the file `somefile.py.lint` in the same directory. That's it, the `pylint` plugin has no other features, it just does its thing.

Important: `pylint` must be installed and accessible by the script for this plugin to work!

javac

The `javac` plugin runs the Java compiler program `javac` on all `.java` files in the repo. Note that it tries to compile *all* files at the same time.

CLI Option

`javac` adds a command line option `-i|--ignore` to `repomate clone`, which takes a space-separated list of files to ignore when compiling.

Configuration

`javac` also adds a configuration file option `ignore` taking a comma-separated list of files, which must be added under the `[javac]` section. Example:

```
[DEFAULTS]
plugins = javac

[javac]
ignore = Main.java, Canvas.java, Other.java
```

Important: The `javac` plugin requires `javac` to be installed and accessible from the command line. All JDK distributions come with `javac`, but you must also ensure that it is on the `PATH` variable.

3.6.4 External Plugins

It's also possible to use plugins that are not included with Repomate. Following the conventions defined in the [repomate-plugin docs](#), all plugins uploaded to PyPi should be named `repomate-<plugin>`, where `<plugin>` is the name of the plugin and thereby the thing to add to the `plugins` option in the configuration file. Any options for the plugin itself should be located under a header named `[<plugin>]`. For example, if I want to use the [repomate-junit4](#) plugin, I first install it:

```
python3 -m pip install repomate-junit4
```

and then use for example this configuration file to activate the plugin, and define some defaults:

```
[DEFAULTS]
plugins = junit4
```

(continues on next page)

(continued from previous page)

```
[junit4]
hamcrest_path = /absolute/path/to/hamcrest-1.3.jar
junit_path = /absolute/path/to/junit-4.12.jar
```

Important: If the configuration file exists, it *must* contain the [DEFAULTS] header, even if you don't put anything in that section. This is to minimize the risk of subtle misconfiguration errors by novice users. If you only want to configure plugins, just add the [DEFAULTS] header by itself, without options, to meet this requirement.

3.7 Migrate master repositories into the target (or master) organization (migrate command)

This step sounds complicated, but it's actually very easy, and can be performed with a single Repomate command. There is however a pre-requisite that must be fulfilled. You must either

- Have local copies of your master repos.

or

- Have all master repos in the same GitHub instance as your target organization.

Assuming we have the repos master-repo-1 and master-repo-2 in the current working directory (i.e. local repos), all we have to do is this:

```
$ repomate migrate -mn master-repo-1 master-repo-2
[INFO] created team master_repos
[INFO] cloning into file:///some/directory/path/master-repo-1
[INFO] cloning into file:///some/directory/path/master-repo-2
[INFO] created repomate-demo/master-repo-1
[INFO] created repomate-demo/master-repo-2
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/master-repo-1 master
[INFO] Pushed files to https://some-enterprise-host/repomate-demo/master-repo-2 master
[INFO] done!
```

Important: If you want to use this command to migrate repos into a master organization, you must specify it with the --org-name option here (instead of the --master-org-name).

There are a few things to note here. First of all, the team master_repos is created. This only happens the first time migrate is run on a new organization. As the name suggests, this team houses all of the master repos. Each master repo that is migrated with the migrate command is added to this team, so they can easily be found at a later time. It may also be confusing that the local repos are being cloned (into a temporary directory). This is simply an implementation detail that does not need much thinking about. Finally, the local repos are pushed to the master branch of the remote repo. This command is perfectly safe to run several times, in case you think you missed something. Running the same thing again yields the following output:

```
$ repomate migrate -mn master-repo-1 master-repo-2
[INFO] cloning into file:///some/directory/path/master-repo-1
[INFO] cloning into file:///some/directory/path/master-repo-2
[INFO] repomate-demo/master-repo-1 already exists
[INFO] repomate-demo/master-repo-2 already exists
```

(continues on next page)

(continued from previous page)

```
[INFO] pushing, attempt 1/3
[INFO] https://some-enterprise-host/repomate-demo/master-repo-1 master is up-to-date
[INFO] https://some-enterprise-host/repomate-demo/master-repo-2 master is up-to-date
[INFO] done!
```

In fact, all Repomate commands that deal with pushing to or cloning from repos in some way are safe to run over and over. This is mostly because of how git works, and has little to do with Repomate itself. Now that our master repos are migrated, we can move on to setting up the student repos!

Note: The `migrate` command can also be used to migrate repos from somewhere on the GitHub instance into the target organization. To do this, use the `-mu` option and provide the urls, instead of `-mn` with local paths. For example, given a repo at `https://some-enterprise-host/other-org/master-repo-1`, it can be migrated into `repomate-demo` by typing

```
$ repomate migrate -mu https://some-enterprise-host/other-org/master-repo-1
```

CHAPTER 4

Configuration

repomate *must* be configured with a mandatory environment variable (see [oauth](#)). Additionally, some of the command line parameters can be pre-configured with e.g. the GitHub instances' API url and the target organization's name (see [config](#)).

Important: The *repomate User Guide* expects there to be a configuration file as described in *Getting started (the show-config, verify-settings and setup commands)*.

4.1 OAUTH token

For repomate to work at all, it needs access to an OAUTH token. See the [GitHub OAUTH docs](#) for how to create a token. Make sure that it has the `repo` and `admin:org` permissions. There are two ways to hand the token to repomate:

1. Put it in the `REPOMATE_OAUTH` environment variable. - On a unix system, this is as simple as `export REPOMATE_OAUTH=<YOUR_TOKEN>`
2. Put it in the configuration file (see [Configuration file](#)).

4.2 Configuration file

An optional configuration file can be added, specifying defaults for several of the most frequently used cli options line options. This is especially useful for teachers and TAs who are managing repos for a single course (and, as a consequence, a single organization).

```
[DEFAULTS]
github_base_url = https://some-api-v3-url
user = YOUR_USERNAME
org_name = ORGANIZATION_NAME
```

(continues on next page)

(continued from previous page)

```
master_org_name = MASTER_ORGANIZATION_NAME
students_file = STUDENTS_FILE_ABSOLUTE_PATH
token = SUPER_SECRET_TOKEN
```

Important: If the configuration file exists, it *must* contain the [DEFAULTS] header. This is to minimize the risk of misconfiguration by novice users.

To find out where to place the configuration file (and what to name it), run `repomate show-config`. The configuration file can also be used to configure `repomate` plugins. See the [Using Existing Plugins](#) section for more details.

Important: Do note that the configuration file contains only default values. Specifying any of the parameters on the command line will override the configuration file's values.

Note: You can run `repomate verify-settings` to verify the basic configuration. This will check the most important settings configurable in DEFAULTS.

CHAPTER 5

CLI documentation

6.1 command

6.2 cli

6.3 config

config module.

Contains the code required for pre-configuring user interfaces.

`repomate.config.check_config_integrity` (*config_file*=`PosixPath('/home/docs/.config/repomate/config.cnf')`)
Raise an exception if the configuration file contains syntactical errors, or if the defaults are misconfigured. Note that plugin options are not checked.

Parameters `config_file` (`Union[str, Path]`) – path to the config file.

Return type `None`

`repomate.config.check_defaults` (*defaults*)

Raise an exception if defaults contain keys that are not configurable arguments.

Parameters `defaults` (`Mapping[str, str]`) – A dictionary of defaults.

`repomate.config.execute_config_hooks` (*config_file*=`PosixPath('/home/docs/.config/repomate/config.cnf')`)

Execute all config hooks.

Parameters `config_file` (`Union[str, Path]`) – path to the config file.

Return type `None`

`repomate.config.get_configured_defaults` (*config_file*=`PosixPath('/home/docs/.config/repomate/config.cnf')`)

Access the config file and return a `ConfigParser` instance with its contents.

Parameters `config_file` (`Union[str, Path]`) – Path to the config file.

Return type `dict`

Returns a dict with the contents of the config file. If there is no config file, the return value is an empty dict.

`repomate.config.get_plugin_names (config_file=PosixPath('/home/docs/.config/repomate/config.cnf'))`

Return a list of unqualified names of plugins listed in the config. The order of the plugins is preserved.

Parameters `config_file` (`Union[str, Path]`) – path to the config file.

Return type `List[str]`

Returns a list of unqualified names of plugin modules, or an empty list if no plugins are listed.

6.4 exception

Modules for all custom repomate exceptions.

All exceptions extend the `RepomateException` base class, which itself extends `Exception`. In other words, exceptions raised within repomate can all be caught by catching `RepomateException`.

exception `repomate.exception.APIError (msg="", *args, **kwargs)`

Raise when something unexpected happens when interacting with the API.

exception `repomate.exception.BadCredentials (msg="", status=None)`

Raise when credentials are rejected.

exception `repomate.exception.CloneFailedError (msg, returncode, stderr, url)`

An error to raise when cloning a repository fails.

exception `repomate.exception.FileError (msg="", *args, **kwargs)`

Raise when reading or writing to a file errors out.

exception `repomate.exception.GitError (msg, returncode, stderr)`

A generic error to raise when a git command exits with a non-zero exit status.

exception `repomate.exception.GitHubError (msg="", status=None)`

An exception raised when the API responds with an error code.

exception `repomate.exception.NotFoundError (msg="", status=None)`

An exception raised when the API responds with a 404.

exception `repomate.exception.ParseError (msg="", *args, **kwargs)`

Raise when something goes wrong in parsing.

exception `repomate.exception.PluginError (msg="", *args, **kwargs)`

Generic error to raise when something goes wrong with loading plugins.

exception `repomate.exception.PushFailedError (msg, returncode, stderr, url)`

An error to raise when pushing to a remote fails.

exception `repomate.exception.RepomateException (msg="", *args, **kwargs)`

Base exception for all repomate exceptions.

exception `repomate.exception.ServiceNotFoundError (msg="", status=None)`

Raise if the base url can't be located.

exception `repomate.exception.UnexpectedException (msg="", status=None)`

An exception raised when an API request raises an unexpected exception.

6.5 github_api

6.6 git

Wrapper functions for git commands.

class `repomate.git.Push` (*local_path*, *repo_url*, *branch*)

branch

Alias for field number 2

local_path

Alias for field number 0

repo_url

Alias for field number 1

`repomate.git.captured_run` (*args, **kwargs)

Run a subprocess and capture the output.

`repomate.git.clone` (*repo_urls*, *token*, *single_branch=True*, *cwd='.'*)

Clone all repos asynchronously.

Parameters

- **repo_urls** (`Iterable[str]`) – URLs to repos to clone.
- **single_branch** (`bool`) – Whether or not to clone only the default branch.
- **cwd** (`str`) – Working directory. Defaults to the current directory.
- **token** (`str`) – A GitHub OAUTH token.

Return type `List[Exception]`

Returns URLs from which cloning failed.

`repomate.git.clone_single` (*repo_url*, *token*, *single_branch=True*, *branch=None*, *cwd='.'*)

Clone a git repository.

Parameters

- **repo_url** (`str`) – HTTPS url to repository on the form `https://<host>/<owner>/<repo>`.
- **single_branch** (`bool`) – Whether or not to clone a single branch.
- **branch** (`Optional[str]`) – The branch to clone.
- **cwd** (`str`) – Working directory. Defaults to the current directory.
- **token** (`str`) – A GitHub OAUTH token.

`repomate.git.push` (*push_tuples*, *user*, *token*, *tries=3*)

Push to all repos defined in *push_tuples* asynchronously. Amount of concurrent tasks is limited by `CONCURRENT_TASKS`. Pushing to repos is tried a maximum of *tries* times (i.e. `pushing is _retried_tries - 1` times.)

Parameters

- **push_tuples** (`Iterable[Push]`) – Push namedtuples defining local and remote repos.
- **user** (`str`) – The username to put in the push.
- **token** (`str`) – A GitHub OAUTH token.

- **tries** (*int*) – Amount of times to try to push (including initial push).

Return type `List[str]`

Returns urls to which pushes failed with exception.PushFailedError. Other errors are only logged.

6.7 tuples

Tuples module.

This module contains various namedtuple containers used throughout repomate. There are still a few namedtuples floating about in their own modules, but the goal is to collect all container types in this module.

```
class repomate.tuples.Args(subparser, org_name, github_base_url, user, master_repo_urls,  
                           master_repo_names, students, issue, title_regex, traceback, state,  
                           show_body, author, num_reviews, master_org_name, token)
```

author

Alias for field number 12

github_base_url

Alias for field number 2

issue

Alias for field number 7

master_org_name

Alias for field number 14

master_repo_names

Alias for field number 5

master_repo_urls

Alias for field number 4

num_reviews

Alias for field number 13

org_name

Alias for field number 1

show_body

Alias for field number 11

state

Alias for field number 10

students

Alias for field number 6

subparser

Alias for field number 0

title_regex

Alias for field number 8

token

Alias for field number 15

traceback

Alias for field number 9

```

user
    Alias for field number 3
class repomate.tuples.Deprecation (replacement, remove_by)

    remove_by
        Alias for field number 1
    replacement
        Alias for field number 0
class repomate.tuples.Issue
class repomate.tuples.Repo
class repomate.tuples.Review (repo, done)

    done
        Alias for field number 1
    repo
        Alias for field number 0
class repomate.tuples.Team (name, members, id)

    id
        Alias for field number 2
    members
        Alias for field number 1
    name
        Alias for field number 0

```

6.8 util

Some general utility functions.

```
repomate.util.find_files_by_extension (root, *extensions)
```

Find all files with the given file extensions, starting from root.

Parameters

- **root** (`Union[str, Path]`) – The directory to start searching.
- **extensions** (`str`) – One or more file extensions to look for.

Return type `Generator[Path, None, None]`

Returns a generator that yields a Path objects to the files.

```
repomate.util.generate_repo_name (team_name, master_repo_name)
```

Construct a repo name for a team.

Parameters

- **team_name** (`str`) – Name of the associated team.
- **master_repo_name** (`str`) – Name of the template repository.

Return type `str`

`repomate.util.generate_repo_names(team_names, master_repo_names)`

Construct all combinations of `generate_repo_name(team_name, master_repo_name)` for the provided team names and master repo names.

Parameters

- **team_names** (`Iterable[str]`) – One or more names of teams.
- **master_repo_names** (`Iterable[str]`) – One or more names of master repositories.

Return type `Iterable[str]`

Returns a list of repo names for all combinations of team and master repo.

`repomate.util.generate_review_team_name(student, master_repo_name)`

Generate a review team name.

Parameters

- **student** (`str`) – A student username.
- **master_repo_name** (`str`) – Name of a master repository.

Return type `str`

Returns a review team name for the student repo associated with this master repo and student.

`repomate.util.is_git_repo(path)`

Check if a directory has a `.git` subdirectory.

Parameters **path** (`str`) – Path to a local directory.

Return type `bool`

Returns True if there is a `.git` subdirectory in the given directory.

`repomate.util.read_issue(issue_path)`

Attempt to read an issue from a textfile. The first line of the file is interpreted as the issue's title.

Parameters **issue_path** (`str`) – Local path to textfile with an issue.

Return type `Issue`

`repomate.util.repo_name(repo_url)`

Extract the name of the repo from its url.

Parameters **repo_url** (`str`) – A url to a repo.

Return type `str`

`repomate.util.validate_non_empty(**kwargs)`

Validate that arguments are not empty. Raise `ValueError` if any argument is empty.

Parameters

- ****kwargs** – Mapping on the form {param_name: argument} where param_name
- **the name of the parameter and argument is the value passed in. (is) –**

Return type `None`

`repomate.util.validate_types(**kwargs)`

Validate argument types. Raise `TypeError` if there is a mismatch.

Parameters

- ****kwargs** – Mapping on the form {param_name: (argument, expected_type)},

- **param_name** is the name of the parameter, argument is the passed (*where*) –
- **value and expected type** is either a single type, or a tuple of (*in*) –
- **types.** –

Return type `None`

6.9 Core plugins

6.9.1 defaults

The defaults plugin contains all default hook implementations.

The goal is to make core parts of repomate pluggable using hooks that only return the first result that is not `None`. The standard behavior will be provided by the default plugin (this one), which implements all of the required hooks. The default plugin will always be run last, so any user-defined hooks will run before it and therefore effectively override the default hooks.

Currently, only the peer review related `generate_review_allocations` hook has a default implementation.

```
repomate.ext.defaults.generate_review_allocations (master_repo_name,          stu-
                                                    dents,          num_reviews,          re-
                                                    view_team_name_function)
```

Generate a (`peer_review_team -> reviewers`) mapping for each student repository (i.e. `<student>-<master_repo_name>`), where `len(reviewers) = num_reviews`.

`review_team_name_function` should be used to generate review team names. It should be called like:

```
review_team_name_function(master_repo_name, student)
```

Important: There must be strictly more students than reviewers per repo (`num_reviews`). Otherwise, allocation is impossible.

Parameters

- **master_repo_name** (`str`) – Name of a master repository.
- **students** (`Iterable[str]`) – Students for which to generate peer review allocations.
- **num_reviews** (`int`) – Amount of reviews each student should perform (and
- **amount of reviewers per repo**) (*consequently*) –
- **review_team_name_function** (`Callable[[str, str], str]`) – A function that takes a master repo name
- **its first argument, and a student username as its second, and** (*as*) –
- **a review team name.** (*returns*) –

Return type `Mapping[str, List[str]]`

Returns a (`peer_review_team -> reviewers`) mapping for each student repository.

6.9.2 pairwise

A peer review plugin which attempts to assign pairwise peer reviews. Intended for students to sit and discuss their code bases with each other, as well as leave feedback. More specifically, N students are split into N/2 groups, each group member assigned to peer review the other person in the group.

If N is odd, the students are split into (N-1)/2 groups, in which one group has 3 members.

```
repomate.ext.pairwise.generate_review_allocations(master_repo_name, students,
                                                    review_team_name_function,
                                                    num_reviews=1)
```

Generate a (peer_review_team -> reviewers) mapping for each student repository (i.e. <student>-<master_repo_name>), where len(reviewers) = 1 or 2.

The num_reviews argument is ignored by this plugin.

Parameters

- **master_repo_name** (*str*) – Name of a master repository.
- **students** (*Iterable[str]*) – Students for which to generate peer review allocations.
- **review_team_name_function** (*Callable[[str, str], str]*) – A function that takes a master repo name as its first argument, and a student username as its second, and returns a review team name.
- **num_reviews** (*int*) – Ignored by this plugin.

Return type *Mapping[str, List[str]]*

Returns a (peer_review_team -> reviewers) mapping for each student repository.

6.10 Extension plugins

6.10.1 javac

6.10.2 pylint

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

c

`config`, [27](#)

d

`defaults`, [33](#)

e

`exception`, [28](#)

g

`git`, [29](#)

p

`pairwise`, [34](#)

r

`repomate.config`, [27](#)

`repomate.exception`, [28](#)

`repomate.ext.defaults`, [33](#)

`repomate.ext.pairwise`, [34](#)

`repomate.git`, [29](#)

`repomate.tuples`, [30](#)

`repomate.util`, [31](#)

t

`tuples`, [30](#)

u

`util`, [31](#)

A

APIError, 28
Args (class in *repomate.tuples*), 30
author (*repomate.tuples.Args* attribute), 30

B

BadCredentials, 28
branch (*repomate.git.Push* attribute), 29

C

captured_run() (in module *repomate.git*), 29
check_config_integrity() (in module *repomate.config*), 27
check_defaults() (in module *repomate.config*), 27
clone() (in module *repomate.git*), 29
clone_single() (in module *repomate.git*), 29
CloneFailedError, 28
config (module), 27

D

defaults (module), 33
Deprecation (class in *repomate.tuples*), 31
done (*repomate.tuples.Review* attribute), 31

E

exception (module), 28
execute_config_hooks() (in module *repomate.config*), 27

F

FileError, 28
find_files_by_extension() (in module *repomate.util*), 31

G

generate_repo_name() (in module *repomate.util*), 31
generate_repo_names() (in module *repomate.util*), 31

generate_review_allocations() (in module *repomate.ext.defaults*), 33
generate_review_allocations() (in module *repomate.ext.pairwise*), 34
generate_review_team_name() (in module *repomate.util*), 32
get_configured_defaults() (in module *repomate.config*), 27
get_plugin_names() (in module *repomate.config*), 28
git (module), 29
GitError, 28
github_base_url (*repomate.tuples.Args* attribute), 30
GitHubError, 28

I

id (*repomate.tuples.Team* attribute), 31
is_git_repo() (in module *repomate.util*), 32
Issue (class in *repomate.tuples*), 31
issue (*repomate.tuples.Args* attribute), 30

L

local_path (*repomate.git.Push* attribute), 29

M

master_org_name (*repomate.tuples.Args* attribute), 30
master_repo_names (*repomate.tuples.Args* attribute), 30
master_repo_urls (*repomate.tuples.Args* attribute), 30
members (*repomate.tuples.Team* attribute), 31

N

name (*repomate.tuples.Team* attribute), 31
NotFoundError, 28
num_reviews (*repomate.tuples.Args* attribute), 30

O

`org_name` (*repomate.tuples.Args* attribute), 30

P

`pairwise` (*module*), 34

`ParseError`, 28

`PluginError`, 28

`Push` (*class in repomate.git*), 29

`push()` (*in module repomate.git*), 29

`PushFailedError`, 28

R

`read_issue()` (*in module repomate.util*), 32

`remove_by` (*repomate.tuples.Deprecation* attribute), 31

`replacement` (*repomate.tuples.Deprecation* attribute), 31

`Repo` (*class in repomate.tuples*), 31

`repo` (*repomate.tuples.Review* attribute), 31

`repo_name()` (*in module repomate.util*), 32

`repo_url` (*repomate.git.Push* attribute), 29

`repomate.config` (*module*), 27

`repomate.exception` (*module*), 28

`repomate.ext.defaults` (*module*), 33

`repomate.ext.pairwise` (*module*), 34

`repomate.git` (*module*), 29

`repomate.tuples` (*module*), 30

`repomate.util` (*module*), 31

`RepomateException`, 28

`Review` (*class in repomate.tuples*), 31

S

`ServiceNotFoundError`, 28

`show_body` (*repomate.tuples.Args* attribute), 30

`state` (*repomate.tuples.Args* attribute), 30

`students` (*repomate.tuples.Args* attribute), 30

`subparser` (*repomate.tuples.Args* attribute), 30

T

`Team` (*class in repomate.tuples*), 31

`title_regex` (*repomate.tuples.Args* attribute), 30

`token` (*repomate.tuples.Args* attribute), 30

`traceback` (*repomate.tuples.Args* attribute), 30

`tuples` (*module*), 30

U

`UnexpectedException`, 28

`user` (*repomate.tuples.Args* attribute), 30

`util` (*module*), 31

V

`validate_non_empty()` (*in module repomate.util*), 32

`validate_types()` (*in module repomate.util*), 32