{cookiecutter.app_name} Documentation

Release 0.12.0

{cookiecutter.author_name}

Dec 30, 2020

Contents:

`repobee-plug` is a package for creating plugins for RepoBee. While the plugin system itself is implemented in the core application, this package contains every part necessary to actually create a plugin. If you just want to use plugins created by others, then you're in the wrong place and should head over to the RepoBee User Guide. If you want to create plugins, you are in precisely the right place! I recommend you read through all of the general documentation, and then refer to the parts of the module reference that you need for your plugin.

Don't hesitate to get in touch if you need help, you can use the issue tracker or find contact details on the official web site.

CHAPTER 1

Plugin system overview

1.1 Conventions

For RepoBee to discover a plugin and its hooks, the following conventions need to be adhered to:

1. The PyPi package should be named `repobee-<plugin>`, where `<plugin>` is the name of the plugin.
2. The actual Python package (i.e. the directory in which the source files are located) should be called `repobee_<plugin>`. In other words, replace the hyphen in the PyPi package name with an underscore.
3. The Python module that defines the plugin's hooks/hook classes should be name `<plugin>.py`.
4. Task plugins that add command line options must prefix the option with `--<plugin>`. So, if the plugin `exampleplug` wants to add the option `--ignore`, then it must be called `--exampleplug-ignore`.
    - The reason for this is to avoid option collisions between different plugins.
    - Note that this does not apply to extension command plugins, as they do not piggyback on existing commands.

For an example plugin that follows these conventions, have a look at repobee-junit4. Granted that the plugin follows these conventions and is installed, it can be loaded like any other RepoBee plugin (see Using Existing Plugins).

1.2 Hooks

There are two types of hooks in RepoBee: *core hooks* and *extension hooks*.

1.2.1 Core hooks

Core hooks provide core functionality for RepoBee, and always have a default implementation in `repobee.ext.defaults`. Providing a different plugin implementation will override this behavior, thereby changing some core part of RepoBee. In general, only one implementation of a core hook will run per invocation of RepoBee. All core hooks are defined in `repobee_plug._corehooks`.

Important: Note that the default implementations in `repobee.ext.defaults` may simply be *imported* into the module. They are not necessarily defined there.

1.2.2 Extension hooks

Extension hooks extend the functionality of RepoBee in various ways. These are probably of most interest to most people looking to create plugins for RepoBee. Unlike the core hooks, there are no default implementations of the extension hooks, and multiple implementations can be run on each invocation of RepoBee. All extension hooks are defined in `repobee_plug._exthooks`.

Tasks

RepoBee has a notion of a *task*, which is a collection of one or more interdependent functions. The purpose of all tasks is to *act* on repositories. For example, the built-in pylint plugin is a task whose act consists of running static analysis on all Python files in a repository. The repobee-junit4 plugin is another task plugin whose act consists of running JUnit4 unit tests on production code in the repository. Tasks can run on master repos before they are pushed to student repos, or on student repos after they have been cloned.

Extension commands

An *extension command* is a top level command that's added to the RepoBee command line interface. The built-in `config-wizard` command is implemented as an extension command, and allows a user of RepoBee to edit the configuration file. The repobee-feedback plugin provides the `issue-feedback` command, which opens feedback issues in student repositories based on local text files. Extension commands are pretty awesome because they integrate seamlessly with RepoBee, can leverage some of RepoBee's powerful CLI functionality and can do pretty much whatever they want on top of that.

1.2.3 Implementing hook functions

There are two ways to implement hooks: as standalone functions or as methods wrapped in a *Plugin* class. In the following two sections, I will briefly show both approaches. For a comprehensive guide on how to use these approaches, refer to the *Creating plugins* section.

Standalone hook functions

Hook functions can be implemented as standalone functions by decorating them with the `repobee_hook()` decorator. For example, if we wanted to implement the `clone_task` hook, we could do it like this:

Listing 1: exampleplug.py

```
import repobee_plug as plug

@plug.repobee_hook
def clone_task():
    """Return a useless Task."""
    return plug.Task(act=act)

def act(path, api):
```

(continues on next page)

(continued from previous page)

```
    return plug.Result(
        name="exampleplug",
        msg="This is a useless plugin!",
        status=plug.Status.SUCCESS,
    )
```

The `clone_task` hook is described in more detail in *Creating plugins*. For a complete plugin written with this approach, see the repobee-gofmt plugin.

Hook functions in a plugin class

Wrapping hook implementations in a class inheriting from *`Plugin`* is recommended way to write plugins for RepoBee that are in any way complicated. A plugin class is instantiated exactly once, and that instance then persists throughout the execution of one RepoBee command, making it a convenient way to implement plugins that require command line options or config values. The *`Plugin`* class also performs some sanity checks when a subclass is defined to make sure that all public functions have hook function names, which comes in handy if you are in the habit of misspelling stuff (aren't we all?). Doing it this way, `exampleplug.py` would look like this:

Listing 2: exampleplug.py

```
import repobee_plug as plug

PLUGIN_NAME = 'exampleplug'

class ExamplePlugin(plug.Plugin):
    """Example plugin that implements the clone_task hook."""

    def clone_task(self):
        """Return a useless Task."""
        return plug.Task(act=self._act)

    def _act(self, path, api):
        return plug.Result(
            name="exampleplug",
            msg="This is a useless plugin!",
            status=plug.Status.SUCCESS,
        )
```

Note how the `clone_task` function now does not have the *@plug.repobee_hook* decorator, that we prefixed `act` with an underscore to signify that it's not a public method (there is no hook function called `act`, so *`Plugin`* will raise if we forget the leading underscore), and that the `self` argument was added to all functions. For a complete example of a plugin written with this approach, see the repobee-junit4 plugin.

CHAPTER 2

Creating plugins

Creating plugins for RepoBee is easy, there is even a template that will start you off with a fully functioning plugin! In this section, I will show you everything you need to know to create task and extension command plugins. Before we begin, you will need to install cookiecutter.

```
$ python3 -m pip install --user --upgrade cookiecutter
```

With this, we will be able to use the repobee-plugin-cookiecutter template to get starter code both for basic and advanced plugins, with minimal effort.

Note: In all of the examples in this tutorial, I will use the plugin name `exampleplug`. This is provided to the template as the `plugin_name` option. Wherever you see `exampleplug` in file names, command line options, configuration files etc, `exampleplug` will be replaced by whatever you provide for the `plugin_name` opiton.

2.1 Creating task plugins

Most plugins for RepoBee are task plugins. The basic idea is that you write some code for doing something (pretty much anything) in a repository, and RepoBee scales your code to operate on any number of student or master repositories. There are currently two types of tasks:

- **Clone task**: operates on student repositories after they have been cloned with the `clone` command.
- **Setup task**: operates on master repositories before they are pushed to student repositories in the `setup` and `update` commands.
  - Currently, a setup task is not allowed to alter the contents of the master Git repository (e.g. with `git commit`), but plans are in motion for allowing this in RepoBee 3.

A task is defined with the *`Task`* data structure, and is more or less just a container for a bunch of callback functions. This allows you as a plugin creator to implement your tasks however you want. Want to just have standalone functions? That's fine. Want to use a class? Also works great.

Whether the task you create is a clone task or a setup task is decided by which hook function(s) you implement. For example, if you implement the `clone_task()` hook to return your task, then you've got a clone task, and if you implement the `setup_task()` hook you've got a setup task. There's no problem implementing both hooks if your task makes sense as both a clone task and a setup task. Let's have a look at a basic task to get an idea for how it works.

2.1.1 Basic

A basic task plugin can be generated with cookiecutter using the `repobee-plugin-cookiecutter` template. Below is a CLI trace of generating one, which you can follow along with. Of course, replace any personal information with your own.

Note: Things such as your name and email are only put into local files (most notably into `setup.py` and `LICENSE`). It's not actually sent anywhere.

Listing 1: Generating a basic task plugin

```
$ python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter
author []: Simon Larsén
email []: slarse@slar.se
github_username []: slarse
plugin_name []: exampleplug
short_description []: An example task plugin
Select generate_basic_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]: 2
Select generate_advanced_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]:
$ ls
repobee-exampleplug
```

After the command has been run, you should have a basic plugin defined locally in the `repobee-exampleplug` directory. Let's have a look at what we got.

```
$ tree repobee-exampleplug
repobee-exampleplug/
├── LICENSE
├── README.md
├── repobee_exampleplug
│   ├── exampleplug.py
│   ├── __init__.py
│   └── __version.py
├── setup.py
└── tests
    └── test_exampleplug.py
```

Note how the directory structure adheres to the conventions defined in *Conventions*. The actual plugin is contained entirely in `repobee_exampleplug/exampleplug.py`, and this is where you want to make changes to alter the behavior of the plugin. Let's have a look at it.

Listing 2: exampleplug.py (note that docstrings have been removed for brevity)

```
import pathlib
import os

import repobee_plug as plug

PLUGIN_NAME = "exampleplug"

def act(path: pathlib.Path, api: plug.API):
    filepaths = [
        str(p) for p in path.resolve().rglob("*") if ".git" not in str(p).split(os.
↪sep)
    ]
    output = os.linesep.join(filepaths)
    return plug.Result(name=PLUGIN_NAME, status=plug.Status.SUCCESS, msg=output)


@plug.repobee_hook
def clone_task() -> plug.Task:
    return plug.Task(act=act)


@plug.repobee_hook
def setup_task() -> plug.Task:
    return plug.Task(act=act)
```

As you can see, it's rather uncomplicated. The `act` function simply finds files in the repository at `path`, and returns a *`Result`* with the results. Returning a *`Result`* is optional, but if you don't RepoBee will not report any results for your plugin. As listing files makes sense both for student and master repos, we can safely implement both the `setup_task` and `clone_task` hooks, and return a *`Task`* with the `act` callback specified. And that's really all there is to to it.

There are some other notable files that you should be familiar with as well.

- `README.md`: You know what this is.
- `LICENSE`: This is the license file, which is relevant if you put this in a public repository (for example on GitHub). It's an MIT license by default, but you can of course change it to whatever you want.
- `setup.py`: This is the file that allows the plugin to be installed. It will work out-of-the-box. If you add any dependencies to your plugin, you must list them in the `required` attribute in `setup.py`. See *Packaging Python Projects <https://packaging.python.org/tutorials/packaging-projects/>* for details.
- `repobee_exampleplug/__version.py`: This contains the version number for the plugin. It defaults to `0.0.1`. This is only important if you plan to distribute your plugin.
- `tests/` A directory with unit tests. It starts with a single default test that makes sure the plugin can be registered with RepoBee, which is a minimum requirement for it actually working.

And that's it for creating a basic plugin.

2.1.2 Interlude - Installing your plugin

Since you're here looking how to create your own plugins, I'm guessing you've already tried using a plugin or two (if not, have a look at the plugin section of the user guide). To be able to use the `exampleplug` plugin that we just created, it needs to be installed. That can easily be done like this:

```
# local install
$ python3 -m pip install --user --upgrade path/to/repobee-exampleplug
# or from a Git repository
$ python3 -m pip install --user --upgrade git+https://urltogitrepo.git
```

Important: Each time you update your plugin, you must install it again!

To check that the plugin was installed correctly and is recognized, we can run RepoBee with the plugin enabled and request the help section.

```
$ repobee -p exampleplug --help
```

In the displayed help section, just over the list of positional arguments, you should see something that looks like this:

```
Loaded plugins: exampleplug-0.0.1, defaults-2.4.0
```

If you see `exampleplug` listed among the plugins, then it was correctly installed! To try it out, you can simply run the `clone` or `setup` command with `exampleplug` enabled. It should give you output like this:

```
$ repobee -p exampleplug clone --mn task-1 -s slarse
[INFO] Cloning into student repos ...
[INFO] Cloned into https://[...]/slarse-task-1
[INFO] Executing tasks ...
[INFO] Processing slarse-task-1
[INFO] hook results for slarse-task-1

exampleplug: SUCCESS
/tmp/tmp_p0v8ha2/slarse-task-1/src
/tmp/tmp_p0v8ha2/slarse-task-1/README.md
/tmp/tmp_p0v8ha2/slarse-task-1/.gitignore
/tmp/tmp_p0v8ha2/slarse-task-1/docs
/tmp/tmp_p0v8ha2/slarse-task-1/src/README.md
/tmp/tmp_p0v8ha2/slarse-task-1/docs/README.md
```

If you've gotten this far, then your plugin is working and you can start adapting it to your needs. If you need more advanced functionality for your task, such as the possibility of providing command line options or config values, then have a look at the advanced task in the next section.

2.1.3 Advanced

You can generate an advanced task plugin with the same cookiecutter template by selecting "yes" on the `generate_advanced_task` option. The advanced task template does the same thing as the basic one, but it also accepts a command line option (`--exampleplug-pattern`), which can also be configured in the config file by adding the `pattern` option to the `[exampleplug]` section. Before you proceed with this section, make sure to have a careful look at the *Task* data structure. When you've done so, proceed with generating a plugin like this:

Listing 3: Generating an advanced task plugin

```
$ python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter
author []: Simon Larsén
email []: slarse@slar.se
github_username []: slarse
plugin_name []: exampleplug
```

(continues on next page)

(continued from previous page)

```
short_description []: An example task plugin
Select generate_basic_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]:
Select generate_advanced_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]: 2
$ ls
repobee-exampleplug
```

The layout will be *exactly* the same as with the *Basic* task, but the `exampleplug.py` file will be much more elaborate. It is a bit on the large side so I won't inline it here, but I can point out the differences.

- The plugin is implemented as a class that extends the *`Plugin`* class, as described in *Hook functions in a plugin class* for non-trivial plugins.
- The `add_option` callback is implemented to add a few options to the parser.
- The `handle_args` callback is also provided to handle the new options added by `add_option`. The reason that `handle_args` is a separate callback, instead of just passing parsed args to the `act` callback, is to allow for fail-fast behavior in case of bad arguments. The `act` callback is typically called fairly late in the execution of RepoBee, but the `handle_args` callback can be called very early.
- It also implements `config_hook()` to access the configuration file. There are a few reasons why there is no `handle_config`-ish callback in *`Task`*. First, config file handling can't depend on the context (e.g. if `setup` or `clone` is called), as the config file is accessed before the CLI arguments are parsed. Second, there are other plugins (such as extension commands) that also need to be able to access the config file, so it's easier to simply have one way of doing it.

Note: If you named your plugin something other than `exampleplug`, then the command line option and config file sections will be named accordingly.

If you install the plugin as specified in the *Interlude - Installing your plugin* section and run `repobee -p exampleplug clone -h`, you should see the added command line option listed in the help section. The plugin can then for example be run like this to list only files ending with `md`:

```
$ repobee -p exampleplug clone --mn task-1 -s slarse --exampleplug-pattern '.*md'
[INFO] Cloning into student repos ...
[INFO] Cloned into https://[...]/slarse-task-1
[INFO] Executing tasks ...
[INFO] Processing slarse-task-1
[INFO] hook results for slarse-task-1

exampleplug: SUCCESS
/tmp/tmp_p0v8ha2/slarse-task-1/README.md
/tmp/tmp_p0v8ha2/slarse-task-1/src/README.md
/tmp/tmp_p0v8ha2/slarse-task-1/docs/README.md
```

That's pretty much it for tasks. Refer to the documentation of the individual parts for details.

2.2 Creating extension command plugins

An extension command is a top-level command in the RepoBee CLI which seamlessly integrates with the base tool. Creating an extension command is fairly similar to creating an advanced task, but it is somewhat easier as an extension command does not need to integrate into an existing command, making the definition simpler. For a user, calling an extension command is as simple as enabling the plugin and running `repobee <EXT_COMMAND_NAME>`. As an example, the built-in `config-wizard` command is actually implemented as an extension command. Before we dive into how to create an extension command plugin, let's first have a look at the core components that make up extension commands.

2.2.1 Extension command components

Extension commands consist of two primary components: the *ExtensionCommand* container and the *ExtensionParser* parser class.

The ExtensionParser

A *ExtensionParser* is fairly straightforward: it's simply a thin wrapper around an `argparse.ArgumentParser` that's instantiated without any arguments. It can then be used identically to an `argparse.ArgumentParser`.

Listing 4: Example usage of an ExtensionParser

```
import repobee_plug as plug

parser = plug.ExtensionParser()
parser.add_argument(
    "-n",
    "--name",
    help="Your name.",
    required=True,
    type=str,
)
parser.add_argument(
    "-a",
    "--age",
    help="Your age.",
    type=int,
)
```

The *ExtensionParser* is then added to an extension command, which we'll have a look at next.

The ExtensionCommand

ExtensionCommand defines an extension command in much the same way as a *Task* defines a task. Most of its properties are self-explanatory, but the `callback`, `requires_api` and `requires_base_parsers` deserve a closer look.

First of all, `requires_base_parsers` is an interesting feature which allows an extension command to request parser components from RepoBee's core parser. The currently available parsers are defined in the `BaseParser` enum. As an example, if you provide `requires_base_parsers=[plug.BaseParser.STUDENTS]`, the `--students` and `--students-file` options are added to the extension parser. Not only does this add options to your parser, but they are processed automatically as well. In the case of the students parser, RepoBee will automatically

check the configuration file for the `students_file` option, and also parse the raw CLI input into a list of *Team* tuples for you. In essence, the parsers you can request to have added are parsed and processed automatically by RepoBee in such a way that your extension command can provide the same experience as RepoBee's core commands, without having to do any work. This is only semi-well documented at the moment, but it's easy enough to simply try passing different base parsers to the `requires_base_parsers`.

The `callback` should be a function that accepts the parsed arguments from the extension command's parser, as well as an *API* instance. Again, if the command requires any base parsers, the arguments from these will be both parsed and processed. The `api` argument is only passed a meaningful value if `requires_api=True`, otherwise `None` is passed.

2.2.2 Basic

Of course, the repobee-plugin-cookiecutter template has starter code for extension commands. There's a basic and an advanced template, and we'll start with the basic one.

Listing 5: Generating a basic extension command plugin

```
$ python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter
author []: Simon Larsén
email []: slarse@slar.se
github_username []: slarse
plugin_name []: exampleplug
short_description []: An example task plugin
Select generate_basic_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]:
Select generate_advanced_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]:
Select generate_basic_extension_command:
1 - no
2 - yes
Choose from 1, 2 [1]: 2
Select generate_advanced_extension_command:
1 - no
2 - yes
Choose from 1, 2 [1]:
$ ls
repobee-exampleplug
```

It will again generate the same directory structure as for tasks, but the plugin will look something like this instead:

Listing 6: exampleplug.py

```
import argparse
import configparser
from typing import List, Mapping, Optional

import repobee_plug as plug

PLUGIN_NAME = "exampleplug"

def callback(
```

(continues on next page)

(continued from previous page)

```
    args: argparse.Namespace, api: Optional[plug.API]
) -> Optional[Mapping[str, List[plug.Result]]]:
    # do whatever you want to do!
    return {
        PLUGIN_NAME: [plug.Result(
            name=PLUGIN_NAME, status=plug.Status.SUCCESS, msg="Hello, world!"
        )]
    }

@plug.repobee_hook
def create_extension_command() -> plug.ExtensionCommand:
    """Create an extension command with no arguments.

    Returns:
        The extension command to add to the RepoBee CLI.
    """
    return plug.ExtensionCommand(
        parser=plug.ExtensionParser(), # empty parser
        name="example-command",
        help="An example command.",
        description="An example extension command.",
        callback=callback,
    )
```

This extension command does nothing, it simply reports some results to RepoBee with the *repobee_plug.Result* data structure. Installing this (see *Interlude - Installing your plugin*) and enabling it (again with `-p exampleplug`) will add the `example-command` command to your RepoBee CLI.

```
$ repobee -p exampleplug example-command
[INFO] hook results for exampleplug

exampleplug: SUCCESS
Hello, world!
```

Not very interesting, but it gives you a base to start on to do very simple extension commands. To also add command line options, configuration file parsing and the like, see the advanced extension.

2.2.3 Advanced

To generate the advanced extension command, simply select it when running the template generation.

Listing 7: Generating an advanced extension command plugin

```
$ python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter
author []: Simon Larsén
email []: slarse@slar.se
github_username []: slarse
plugin_name []: exampleplug
short_description []: An example task plugin
Select generate_basic_task:
1 - no
2 - yes
Choose from 1, 2 (1, 2) [1]:
Select generate_advanced_task:
1 - no
```

(continues on next page)

(continued from previous page)

```
2 - yes
Choose from 1, 2 (1, 2) [1]:
Select generate_basic_extension_command:
1 - no
2 - yes
Choose from 1, 2 [1]:
Select generate_advanced_extension_command:
1 - no
2 - yes
Choose from 1, 2 [1]: 2
$ ls
repobee-exampleplug
```

Again, it will have the exact same directory structure as all the other plugins that we've generated, and all differences are contained in `exampleplug.py`. This extension command adds options, uses the configuration file and has internal state. It is much too large to include here, but I recommend that you simply read the source code and try to figure out how it works. Given the time, I will add more elaborate instructions here, but right now this is as far as I can take it.

CHAPTER 3

Public API

All of the public functions and classes of `repobee-plug` can be imported from the top-level package `repobee_plug`. Only this API is stable, the internal modules currently are not. The recommended way to use `repobee_plug` is to import it either as is, or alias it to `plug`. Typically, I do the latter.

Listing 1: Example usage of repobee-plug

```
import repobee_plug as plug

def act(path, api):
    result = plug.Result(
        name="hello",
        msg="Hello, world!",
        status=plug.Status.SUCCESS,
    )
    return result

@plug.repobee_hook
def clone_task():
    """A hello world clone task."""
    return plug.Task(act=act)
```

3.1 Hook functions

There are two parts to hook functions in RepoBee: the specifications of the hook functions, and the implementation markers to signify that you have (attempted) to implement a hook.

3.1.1 Implementation markers

There are two ways to mark a function as a hook implementation: with the `repobee_hook` decorator or using the `Plugin` class.

class repobee_plug.**Plugin**

This is a base class for plugin classes. For plugin classes to be picked up by RepoBee, they must inherit from this class.

Public methods must be hook methods. If there are any public methods that are not hook methods, an error is raised on creation of the class. As long as the method has the correct name, it will be recognized as a hook method during creation. However, if the signature is incorrect, the plugin framework will raise a runtime exception once it is called. Private methods (i.e. methods prefixed with _) carry no restrictions.

The signatures of hook methods are not checked until the plugin class is registered by the `repobee_plug.manager` (an instance of `pluggy.manager.PluginManager`). Therefore, when testing a plugin, it is a good idea to include a test where it is registered with the manager to ensure that it has the correct signatures.

A plugin class is instantiated exactly once; when RepoBee loads the plugin. This means that any state that is stored in the plugin will be carried throughout the execution of a RepoBee command. This makes plugin classes well suited for implementing tasks that require command line options or configuration values, as well as for implementing extension commands.

3.1.2 Extension hooks

Important: The hook function specifications are part of the public API for documentation purposes only. You should not import or use these function in any way, but only implement them as described in *Implementing hook functions*.

Hookspecs for repobee extension hooks.

Extension hooks add something to the functionality of repobee, but are not necessary for its operation. Currently, all extension hooks are related to cloning repos.

class repobee_plug._exthooks.**CloneHook**

Hook functions related to cloning repos.

act_on_cloned_repo(*path*, *api*)

Do something with a cloned repo.

Deprecated since version 0.12.0: This hook is has been replaced by `TaskHooks.clone_task()`. Once all known, existing plugins have been migrated to the new hook, this hook will be removed.

Parameters

- **path** (`Union[str, Path]`) – Path to the repo.
- **api** (`API`) – An instance of `repobee.github_api.GitHubAPI`.

Return type `Optional[Result]`

Returns optionally returns a Result namedtuple for reporting the outcome of the hook. May also return None, in which case no reporting will be performed for the hook.

clone_parser_hook(*clone_parser*)

Do something with the clone repos subparser before it is used used to parse CLI options. The typical task is to add options to it.

Deprecated since version 0.12.0: This hook is has been replaced by `TaskHooks.clone_task()`. Once all known, existing plugins have been migrated to the new hook, this hook will be removed.

Parameters **clone_parser** (`ArgumentParser`) – The `clone` subparser.

Return type `None`

config_hook(*config_parser*)
Hook into the config file parsing.

Parameters **config** – the config parser after config has been read.

Return type None

parse_args(*args*)
Get the raw args from the parser. Only called for the clone parser. The typical task is to fetch any values from options added in clone_parser_hook().

Parameters **args** (Namespace) – The full namespace returned by argparse.ArgumentParser.parse_args()

Return type None

class repobee_plug._exthooks.**ExtensionCommandHook**
Hooks related to extension commands.

create_extension_command()
Create an extension command to add to the RepoBee CLI. The command will be added as one of the top-level subcommands of RepoBee. This hook is called precisely once, and should return an *ExtensionCommand*.

```
def command(args: argparse.Namespace, api: apimeta.API)
```

The command function will be called if the extension command is used on the command line.

Note that the RepoBeeExtensionParser class is just a thin wrapper around argparse.ArgumentParser, and can be used in an identical manner. The following is an example definition of this hook that adds a subcommand called example-command, that can be called with repobee example-command.

```
import repobee_plug as plug

def callback(args: argparse.Namespace, api: plug.API) -> None:
    LOGGER.info("callback called with: {}, {}".format(args, api))

@plug.repobee_hook
def create_extension_command():
    parser = plug.RepoBeeExtensionParser()
    parser.add_argument("-b", "--bb", help="A useless argument")
    return plug.ExtensionCommand(
        parser=parser,
        name="example-command",
        help="An example command",
        description="Description of an example command",
        callback=callback,
    )
```

Important: If you need to use the api, you set requires_api=True in the ExtensionCommand. This will automatically add the options that the API requires to the CLI options of the subcommand, and initialize the api and pass it in.

See the documentation for *ExtensionCommand* for more details on it.

Return type ExtensionCommand

Returns A ExtensionCommand.

class repobee_plug._exthooks.**TaskHooks**
Hook functions relating to RepoBee tasks.

clone_task()
Create a task to run on a copy of a cloned student repo. This hook replaces the old `act_on_cloned_repo` hook.

Implementations of this hook should return a `Task`, which defines a callback that is called after all student repos have been cloned. See the definition of `Task` for details.

Return type `Task`

Returns A `Task` instance defining a RepoBee task.

setup_task()
Create a task to run on a copy of the master repo before it is pushed out to student repositories. This can for example be pre-flight checks of the master repo, or something else entirely.

Implementations of this hook should return a `Task`, which defines a callback that is called after the master repo has been safely copied, but before that copy is pushed out to student repositories. Note that any changes to the repository must be committed to actually show up in the student repositories.

Note: Structural changes to the master repo are not currently supported. Changes to the repository during the callback will not be reflected in the generated repositories. Support for preprocessing (such that changes do take effect) is a potential future feature.

Return type `Task`

3.1.3 Core hooks

Important: The hook function specifications are part of the public API for documentation purposes only. You should not import or use these function in any way, but only implement them as described in *Implementing hook functions*.

Hookspecs for repobee core hooks.

Core hooks provide the basic functionality of repobee. These hooks all have default implementations, but are overridden by any other implementation. All hooks in this module should have the *firstresult=True* option to the hookspec to allow for this dynamic override.

class repobee_plug._corehooks.**APIHook**
Hooks related to platform APIs.

api_init_requires()
Return which of the arguments to apimeta.APISpec.__init__ that the given API requires. For example, the GitHubAPI requires all, but the GitLabAPI does not require `user`.

Return type `Tuple[str]`

Returns Names of the required arguments.

get_api_class()
Return an API platform class. Must be a subclass of apimeta.API.

Returns An apimeta.API subclass.

class repobee_plug._corehooks.**PeerReviewHook**
Hook functions related to allocating peer reviews.

generate_review_allocations(*teams*, *num_reviews*)

Generate `ReviewAllocation` tuples from the provided teams, given that this concerns reviews for a single master repo.

The provided teams of students should be treated as units. That is to say, if there are multiple members in a team, they should always be assigned to the same review team. The best way to merge two teams `team_a` and `team_b` into one review team is to simply do:

```
team_c = apimeta.Team(members=team_a.members + team_b.members)
```

This can be scaled to however many teams you would like to merge. As a practical example, if teams `team_a` and `team_b` are to review `team_c`, then the following `ReviewAllocation` tuple, here called `allocation`, should be contained in the returned list.

```
review_team = apimeta.Team(members=team_a.members + team_b.members)
allocation = containers.ReviewAllocation(
    review_team=review_team,
    reviewed_team=team_c,
)
```

Note: Respecting the `num_reviews` argument is optional: only do it if it makes sense. It's good practice to issue a warning if num_reviews is ignored, however.

Parameters

- **team** – A list of `Team` tuples.
- **num_reviews** (`int`) – Amount of reviews each student should perform (and consequently amount of reviewers per repo)

Return type `List[ReviewAllocation]`

Returns

A list of **ReviewAllocation** tuples.

3.2 API Wrappers

The API wrappers in `repobee-plug` provide a level of abstraction from the the underlying platform API (e.g. GitHub or GitLab), and allows RepoBee to work with different platforms. To fully support a new platform, the *API* must be subclassed an all of its functions implemented. It is possible to support a subset of the functionality as well, but you will need to look into the RepoBee implementation to see which API methods are required for which commands.

class repobee_plug.**Team**

Wrapper class for a Team API object.

class repobee_plug.**TeamPermission**

Enum specifying team permissions on creating teams. On GitHub, for example, this can be e.g. *push* or *pull*.

class repobee_plug.**Issue**

Wrapper class for an Issue API object.

static from_dict(*asdict*)

Take a dictionary produced by Issue.to_dict and reconstruct the corresponding instance. The `implementation` field is lost in a to_dict -> from_dict roundtrip.

Return type `Issue`

to_dict()
Return a dictionary representation of this namedtuple, without the `implementation` field.

class repobee_plug.**IssueState**
Enum specifying a possible issue state.

class repobee_plug.**Repo**
Wrapper class for a Repo API object.

class repobee_plug.**API**(*base_url*, *token*, *org_name*, *user*)
API base class that all API implementations should inherit from. This class functions similarly to an abstract base class, but with a few key distinctions that affect the inheriting class.

1. Public methods *must* override one of the public methods of `APISpec`. If an inheriting class defines any other public method, an `APIError` is raised when the class is defined.
2. All public methods in `APISpec` have a default implementation that simply raise a `NotImplementedError`. There is no requirement to implement any of them.

add_repos_to_review_teams(*team_to_repos*, *issue=None*)
Add repos to review teams. For each repo, an issue is opened, and every user in the review team is assigned to it. If no issue is specified, sensible defaults for title and body are used.

Parameters

- **team_to_repos** (`Mapping[str, Iterable[str]]`) – A mapping from a team name to an iterable of repo names.
- **issue** (`Optional[Issue]`) – An optional Issue tuple to override the default issue.

Return type `None`

close_issue(*title_regex*, *repo_names*)
Close any issues in the given repos in the target organization, whose titles match the title_regex.

Parameters

- **title_regex** (`str`) – A regex to match against issue titles.
- **repo_names** (`Iterable[str]`) – Names of repositories to close issues in.

Return type `None`

create_repos(*repos*)
Create repos in the target organization according the those specced by the `repos` argument. Repos that already exist are skipped.

Parameters **repos** (`Iterable[Repo]`) – Repos to be created.

Return type `List[str]`

Returns A list of urls to the repos specified by the `repos` argument, both those that were created and those that already existed.

delete_teams(*team_names*)
Delete all teams in the target organizatoin that exactly match one of the provided `team_names`. Skip any team name for which no match is found.

Parameters **team_names** (`Iterable[str]`) – A list of team names for teams to be deleted.

Return type `None`

discover_repos(*teams*)
Return all repositories related to the provided teams.

Parameters **teams** (`Iterable[Team]`) – Team tuples.

Return type Generator[Repo, None, None]

Returns A list of Repo tuples.

ensure_teams_and_members(*teams*, *permission=<TeamPermission.PUSH: 'push'>*)
Ensure that the teams exist, and that their members are added to the teams.

Teams that do not exist are created, teams that already exist are fetched. Members that are not in their teams are added, members that do not exist or are already in their teams are skipped.

Parameters

- **teams** (Iterable[Team]) – A list of teams specifying student groups.
- **permission** (TeamPermission) – The permission these teams (or members of them) should be given in regards to associated repositories.

Return type List[Team]

Returns A list of Team API objects of the teams provided to the function, both those that were created and those that already existed.

extract_repo_name(*repo_url*)
Extract a repo name from the provided url.

Parameters **repo_url** (str) – A URL to a repository.

Return type str

Returns The name of the repository corresponding to the url.

get_issues(*repo_names*, *state=<IssueState.OPEN: 'open'>*, *title_regex=''*)
Get all issues for the repos in repo_names an return a generator that yields (repo_name, issue generator) tuples. Will by default only get open issues.

Parameters

- **repo_names** (Iterable[str]) – An iterable of repo names.
- **state** (IssueState) – Specifies the state of the issue.
- **title_regex** (str) – If specified, only issues matching this regex are
- **Defaults to the empty string**(*returned.*) –

Return type Generator[Tuple[str, Generator[Issue, None, None]], None, None]

Returns A generator that yields (repo_name, issue_generator) tuples.

get_repo_urls(*master_repo_names*, *org_name=None*, *teams=None*)
Get repo urls for all specified repo names in the organization. As checking if every single repo actually exists takes a long time with a typical REST API, this function does not in general guarantee that the urls returned actually correspond to existing repos.

If the org_name argument is supplied, urls are computed relative to that organization. If it is not supplied, the target organization is used.

If the *teams* argument is supplied, student repo urls are computed instead of master repo urls.

Parameters

- **master_repo_names** (Iterable[str]) – A list of master repository names.
- **org_name** (Optional[str]) – Organization in which repos are expected. Defaults to the target organization of the API instance.

- **teams** (Optional[List[Team]]) – A list of teams specifying student groups. Defaults to None.

Return type List[str]

Returns a list of urls corresponding to the repo names.

get_review_progress(*review_team_names*, *teams*, *title_regex*)

Get the peer review progress for the specified review teams and student teams by checking which review team members have opened issues in their assigned repos. Only issues matching the title regex will be considered peer review issues. If a reviewer has opened an issue in the assigned repo with a title matching the regex, the review will be considered done.

Note that reviews only count if the student is in the review team for that repo. Review teams must only have one associated repo, or the repo is skipped.

Parameters

- **review_team_names** (Iterable[str]) – Names of review teams.
- **teams** (Iterable[Team]) – Team API objects specifying student groups.
- **title_regex** (str) – If an issue title matches this regex, the issue is considered a potential peer review issue.

Return type Mapping[str, List]

Returns a mapping (reviewer -> assigned_repos), where reviewer is a str and assigned_repos is a _repobee.tuples.Review.

get_teams()

Get all teams related to the target organization.

Return type List[Team]

Returns A list of Team API object.

open_issue(*title*, *body*, *repo_names*)

Open the specified issue in all repos with the given names, in the target organization.

Parameters

- **title** (str) – Title of the issue.
- **body** (str) – Body of the issue.
- **repo_names** (Iterable[str]) – Names of repos to open the issue in.

Return type None

static verify_settings(*user*, *org_name*, *base_url*, *token*, *master_org_name=None*)

Verify the following (to the extent that is possible and makes sense for the specifi platform):

1. Base url is correct
2. The token has sufficient access privileges
3. **Target organization (specifiend by org_name) exists**
   - If master_org_name is supplied, this is also checked to exist.
4. **User is owner in organization (verify by getting**
   - If master_org_name is supplied, user is also checked to be an owner of it.

organization member list and checking roles)

Should raise an appropriate subclass of `_repobee.exception.APIError` when a problem is encountered.

Parameters

- **user** (`str`) – The username to try to fetch.
- **org_name** (`str`) – Name of the target organization.
- **base_url** (`str`) – A base url to a github API.
- **token** (`str`) – A secure OAUTH2 token.
- **org_name** – Name of the master organization.

Returns True if the connection is well formed.

Raises `_repobee.exception.APIError`

3.3 Containers

The containers in `repobee-plug` are immutable classes for storing data. Probably the most important containers are the *`Result`* and the *`Task`* classes.

class repobee_plug.**Result**(*name*, *status*, *msg*, *data=None*)

Container for storing results from hooks.

Parameters

- **name** (`str`) – Name to associate with this result. This is typically the name of the plugin that returns this result.
- **status** (`Status`) – Status of the plugin execution.
- **msg** (`str`) – A free-form result message.
- **data** (`Optional[Mapping[Any, Any]]`) – Semi-structured data in the form of a dictionary. All of the contents of the dictionary should be serializable as this is primarily used for JSON storage.

class repobee_plug.**Task**(*act*, *add_option=None*, *handle_args=None*, *persist_changes=False*)

A data structure for describing a task. Tasks are operations that plugins can define to run on for example cloned student repos (a clone task) or on master repos before setting up student repos (a setup task). Only the `act` attribute is required, all other attributes can be omitted.

The callback methods should have the following headers.

```
def act(
    path: pathlib.Path, api: repobee_plug.API
) -> Optional[containers.Result]:

def add_option(parser: argparse.ArgumentParser) -> None:

def handle_args(args: argparse.Namespace) -> None:
```

Note: The functions are called in the following order: `add_option` -> `handle_args` -> `act`.

Important: The `act` callback should *never* change the Git repository it acts upon (e.g. running commands such as `git add`, `git checkout` or `git commit`). This can have adverse and unexpected effects on RepoBee's functionality. It is however absolutely fine to change the files in the Git working tree, as long as nothing is added or committed.

Each callback is called at most once. They are not guaranteed to execute, because there may be an unexpected crash somewhere else, or the plugin may not come into scope (for example, a clone task plugin will not come into scope if `repobee setup` is run). The callbacks can do whatever is appropriate for the plugin, except for changing any Git repositories. For information on the types used in the callbacks, see the Python stdlib documentation for `argparse`.

As an example, a simple clone task can be defined like so:

```
import repobee_plug as plug

def act(path, api):
    return plug.Result(
        name="example",
        msg="IT LIVES!",
        status=plug.Status.SUCCESS
    )

@plug.repobee_hook
def clone_task():
    return plug.Task(act=act)
```

If your task plugin also needs to access the configuration file, then implement the separate `config_hook` hook. For more elaborate instructions on creating tasks, see the tutorial.

Parameters

- **act** (`Callable`[[`Path`, `API`], `Result`]) – A required callback function that takes the path to a repository worktree and an API instance, and optionally returns a Result to report results.
- **add_option** (`Optional`[`Callable`[[`ArgumentParser`], `None`]]) – An optional callback function that adds options to the CLI parser.
- **handle_args** (`Optional`[`Callable`[[`Namespace`], `None`]]) – An optional callback function that receives the parsed CLI args.
- **persist_changes** (`bool`) – If True, the task requires that changes to the repository that has been acted upon be persisted. This means different things in different contexts (e.g. whether the task is executed in a clone context or in a setup context), and may not be supported for all contexts.

class repobee_plug.**Deprecation**

Parameters

- **replacement** (*str*) – The functionality that replaces the deprecated functionality.
- **remove_by_version** (*str*) – A version number on the form `MAJOR.MINOR.PATCH` by which the deprecated functionality will be removed.

Create new instance of Deprecation(replacement, remove_by_version)

remove_by_version
Alias for field number 1

replacement
Alias for field number 0

class repobee_plug.**Status**
Status codes enums for Results.

SUCCESS
Signifies a plugin execution without any complications.

WARNING
Signifies a plugin execution with non-critical failures.

ERROR
Signifies a critical error during execution.

class repobee_plug.**ExtensionCommand**(*parser*, *name*, *help*, *description*, *callback*, *requires_api=False*, *requires_base_parsers=None*)
Class defining an extension command for the RepoBee CLI.

Parameters

- **parser** (ExtensionParser) – The parser to use for the CLI.
- **name** (str) – Name of the command.
- **help** (str) – Text that will be displayed when running repobee -h
- **description** (str) – Text that will be displayed when calling the -h option for this specific command. Should be elaborate in describing the usage of the command.
- **callback** (Callable[[Namespace, Optional[API]], Optional[Mapping[str, Result]]]) – A callback function that is called if this command is used on the CLI. It is passed the parsed namespace and the platform API. It may optionally return a result mapping on the form (name: str -> List[Result]) that's reported by RepoBee.
- **requires_api** (bool) – If True, the base arguments required for the platform API are added as options to the extension command, and the platform API is then passed to the callback function. It is then important not to have clashing option names. If False, the base arguments are not added to the CLI, and None is passed in place of the API. If you include REPO_DISCOVERY in requires_base_parsers, then you *must* set this to True.
- **requires_base_parsers** (Optional[Iterable[BaseParser]]) – A list of repobee_plug.BaseParser that decide which base parsers are added to this command. For example, setting requires_base_parsers = [BaseParser.STUDENTS] adds the --students and --students-file options to this extension command's parser.

class repobee_plug.**ExtensionParser**
An ArgumentParser specialized for RepoBee extension commands.

add_argument(*dest*, ..., *name=value*, ...)
add_argument(option_string, option_string, ..., name=value, ...)

error(*message: string*)
Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

class repobee_plug.**ReviewAllocation**

Parameters

- **review_team** (Team) – The team of reviewers.
- **reviewed_team** (Team) – The team that is to be reviewed.

Create new instance of ReviewAllocation(review_team, reviewed_team)

review_team
Alias for field number 0

reviewed_team
Alias for field number 1

class repobee_plug.**Review**

Parameters

- **repo** (Repo) – The reviewed repository.
- **done** (*bool*) – Whether or not the review is done.

Create new instance of Review(repo, done)

done
Alias for field number 1

repo
Alias for field number 0

3.4 Helpers

repobee-plug defines various helper functions and classes for use in both RepoBee core and in plugins. These vary from generating repo names, to handling deprecation, to mapping key data structures from and to JSON.

repobee_plug.**json_to_result_mapping**(*json_string*)
Deserialize a JSON string to a mapping repo_name: str -> hook_results: List[Result]

Return type Mapping[str, List[Result]]

repobee_plug.**result_mapping_to_json**(*result_mapping*)
Serialize a result mapping repo_name: str -> hook_results: List[Result] to JSON.

Return type str

repobee_plug.**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

repobee_plug.**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.

repobee_plug.**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.

repobee_plug.**deprecated_hooks**()

Return type Mapping[str, Deprecation]

Returns A mapping of hook names to Deprecation tuples.

3.5 Exceptions

exception repobee_plug.**PlugError**(**args*, ***kwargs*)
Base class for all repobee_plug exceptions.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to Exception. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument a caused the error, then you should pass a=a as a kwarg so it can be introspected at a later time.

exception repobee_plug.**ExtensionCommandError**(**args*, ***kwargs*)
Raise when an :py:class:~repobee_plug.containers.ExtensionCommand: is incorrectly defined.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to Exception. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument a caused the error, then you should pass a=a as a kwarg so it can be introspected at a later time.

exception repobee_plug.**HookNameError**(**args*, ***kwargs*)
Raise when a public method in a class that inherits from *Plugin* does not have a hook name.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to Exception. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument a caused the error, then you should pass a=a as a kwarg so it can be introspected at a later time.

CHAPTER 4

repobee_plug Module Reference

This is the internal API of repobee_plug, which is not stable.

4.1 _apimeta

Metaclass for API implementations.

APIMeta defines the behavior required of platform API implementations, based on the methods in APISpec. With platform API, we mean for example the GitHub REST API, and the GitLab REST API. The point is to introduce another layer of indirection such that higher levels of RepoBee can use different platforms in a platform-independent way. API is a convenience class so consumers don't have to use the metaclass directly.

Any class implementing a platform API should derive from API. It will enforce that all public methods are one of the method defined py APISpec, and give a default implementation (that just raises NotImplementedError) for any unimplemented API methods.

class repobee_plug._apimeta.**API**(*base_url*, *token*, *org_name*, *user*)
API base class that all API implementations should inherit from. This class functions similarly to an abstract base class, but with a few key distinctions that affect the inheriting class.

1. Public methods *must* override one of the public methods of APISpec. If an inheriting class defines any other public method, an APIError is raised when the class is defined.
2. All public methods in APISpec have a default implementation that simply raise a NotImplementedError. There is no requirement to implement any of them.

class repobee_plug._apimeta.**APIMeta**
Metaclass for an API implementation. All public methods must be a specified api method, but all api methods do not need to be implemented.

class repobee_plug._apimeta.**APIObject**
Base wrapper class for platform API objects.

class repobee_plug._apimeta.**APISpec**(*base_url*, *token*, *org_name*, *user*)
Wrapper class for API method stubs.

Important: This class should not be inherited from directly, it serves only to document the behavior of a platform API. Classes that implement this behavior should inherit from `API`.

add_repos_to_review_teams(*team_to_repos*, *issue=None*)
Add repos to review teams. For each repo, an issue is opened, and every user in the review team is assigned to it. If no issue is specified, sensible defaults for title and body are used.

Parameters

- **team_to_repos** (`Mapping[str, Iterable[str]]`) – A mapping from a team name to an iterable of repo names.
- **issue** (`Optional[Issue]`) – An optional Issue tuple to override the default issue.

Return type `None`

close_issue(*title_regex*, *repo_names*)
Close any issues in the given repos in the target organization, whose titles match the title_regex.

Parameters

- **title_regex** (`str`) – A regex to match against issue titles.
- **repo_names** (`Iterable[str]`) – Names of repositories to close issues in.

Return type `None`

create_repos(*repos*)
Create repos in the target organization according the those specced by the `repos` argument. Repos that already exist are skipped.

Parameters **repos** (`Iterable[Repo]`) – Repos to be created.

Return type `List[str]`

Returns A list of urls to the repos specified by the `repos` argument, both those that were created and those that already existed.

delete_teams(*team_names*)
Delete all teams in the target organizatoin that exactly match one of the provided `team_names`. Skip any team name for which no match is found.

Parameters **team_names** (`Iterable[str]`) – A list of team names for teams to be deleted.

Return type `None`

discover_repos(*teams*)
Return all repositories related to the provided teams.

Parameters **teams** (`Iterable[Team]`) – Team tuples.

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

Returns A list of Repo tuples.

ensure_teams_and_members(*teams*, *permission=<TeamPermission.PUSH: 'push'>*)
Ensure that the teams exist, and that their members are added to the teams.

Teams that do not exist are created, teams that already exist are fetched. Members that are not in their teams are added, members that do not exist or are already in their teams are skipped.

Parameters

- **teams** (`Iterable[Team]`) – A list of teams specifying student groups.

- **permission** (TeamPermission) – The permission these teams (or members of them) should be given in regards to associated repositories.

Return type List[Team]

Returns A list of Team API objects of the teams provided to the function, both those that were created and those that already existed.

extract_repo_name(*repo_url*)
Extract a repo name from the provided url.

Parameters **repo_url** (str) – A URL to a repository.

Return type str

Returns The name of the repository corresponding to the url.

get_issues(*repo_names*, *state=<IssueState.OPEN: 'open'>*, *title_regex=''*)
Get all issues for the repos in repo_names an return a generator that yields (repo_name, issue generator) tuples. Will by default only get open issues.

Parameters

- **repo_names** (Iterable[str]) – An iterable of repo names.
- **state** (IssueState) – Specifies the state of the issue.
- **title_regex** (str) – If specified, only issues matching this regex are
- **Defaults to the empty string**(*returned.*) –

Return type Generator[Tuple[str, Generator[Issue, None, None]], None, None]

Returns A generator that yields (repo_name, issue_generator) tuples.

get_repo_urls(*master_repo_names*, *org_name=None*, *teams=None*)
Get repo urls for all specified repo names in the organization. As checking if every single repo actually exists takes a long time with a typical REST API, this function does not in general guarantee that the urls returned actually correspond to existing repos.

If the org_name argument is supplied, urls are computed relative to that organization. If it is not supplied, the target organization is used.

If the *teams* argument is supplied, student repo urls are computed instead of master repo urls.

Parameters

- **master_repo_names** (Iterable[str]) – A list of master repository names.
- **org_name** (Optional[str]) – Organization in which repos are expected. Defaults to the target organization of the API instance.
- **teams** (Optional[List[Team]]) – A list of teams specifying student groups. Defaults to None.

Return type List[str]

Returns a list of urls corresponding to the repo names.

get_review_progress(*review_team_names*, *teams*, *title_regex*)
Get the peer review progress for the specified review teams and student teams by checking which review team members have opened issues in their assigned repos. Only issues matching the title regex will be considered peer review issues. If a reviewer has opened an issue in the assigned repo with a title matching the regex, the review will be considered done.

Note that reviews only count if the student is in the review team for that repo. Review teams must only have one associated repo, or the repo is skipped.

Parameters

- **review_team_names** (Iterable[str]) – Names of review teams.
- **teams** (Iterable[Team]) – Team API objects specifying student groups.
- **title_regex** (str) – If an issue title matches this regex, the issue is considered a potential peer review issue.

Return type Mapping[str, List]

Returns a mapping (reviewer -> assigned_repos), where reviewer is a str and assigned_repos is a _repobee.tuples.Review.

get_teams()

Get all teams related to the target organization.

Return type List[Team]

Returns A list of Team API object.

open_issue(*title*, *body*, *repo_names*)

Open the specified issue in all repos with the given names, in the target organization.

Parameters

- **title** (str) – Title of the issue.
- **body** (str) – Body of the issue.
- **repo_names** (Iterable[str]) – Names of repos to open the issue in.

Return type None

static verify_settings(*user*, *org_name*, *base_url*, *token*, *master_org_name=None*)

Verify the following (to the extent that is possible and makes sense for the specifi platform):

1. Base url is correct
2. The token has sufficient access privileges
3. **Target organization (specifiend by org_name) exists**
   - If master_org_name is supplied, this is also checked to exist.
4. **User is owner in organization (verify by getting**
   - If master_org_name is supplied, user is also checked to be an owner of it.

organization member list and checking roles)

Should raise an appropriate subclass of _repobee.exception.APIError when a problem is encountered.

Parameters

- **user** (str) – The username to try to fetch.
- **org_name** (str) – Name of the target organization.
- **base_url** (str) – A base url to a github API.
- **token** (str) – A secure OAUTH2 token.
- **org_name** – Name of the master organization.

Returns True if the connection is well formed.

Raises _repobee.exception.APIError

class repobee_plug._apimeta.**Issue**
Wrapper class for an Issue API object.

static from_dict(*asdict*)
Take a dictionary produced by Issue.to_dict and reconstruct the corresponding instance. The implementation field is lost in a to_dict -> from_dict roundtrip.

Return type Issue

to_dict()
Return a dictionary representation of this namedtuple, without the implementation field.

class repobee_plug._apimeta.**IssueState**
Enum specifying a possible issue state.

class repobee_plug._apimeta.**Repo**
Wrapper class for a Repo API object.

class repobee_plug._apimeta.**Team**
Wrapper class for a Team API object.

class repobee_plug._apimeta.**TeamPermission**
Enum specifying team permissions on creating teams. On GitHub, for example, this can be e.g. *push* or *pull*.

repobee_plug._apimeta.**check_init_params**(*reference_params*, *compare_params*)
Check that the compare __init__'s parameters are a subset of the reference class's version.

repobee_plug._apimeta.**check_parameters**(*reference*, *compare*)
Check if the parameters match, one by one. Stop at the first diff and raise an exception for that parameter.

An exception is made for __init__, for which the compare may be a subset of the reference in no particular order.

repobee_plug._apimeta.**methods**(*attrdict*)
Return all public methods and __init__ for some class.

repobee_plug._apimeta.**parameters**(*function*)
Extract parameter names and default arguments from a function.

4.2 _pluginmeta

class repobee_plug._pluginmeta.**Plugin**
This is a base class for plugin classes. For plugin classes to be picked up by RepoBee, they must inherit from this class.

Public methods must be hook methods. If there are any public methods that are not hook methods, an error is raised on creation of the class. As long as the method has the correct name, it will be recognized as a hook method during creation. However, if the signature is incorrect, the plugin framework will raise a runtime exception once it is called. Private methods (i.e. methods prefixed with _) carry no restrictions.

The signatures of hook methods are not checked until the plugin class is registered by the repobee_plug.manager (an instance of pluggy.manager.PluginManager). Therefore, when testing a plugin, it is a good idea to include a test where it is registered with the manager to ensure that it has the correct signatures.

A plugin class is instantiated exactly once; when RepoBee loads the plugin. This means that any state that is stored in the plugin will be carried throughout the execution of a RepoBee command. This makes plugin classes

well suited for implementing tasks that require command line options or configuration values, as well as for implementing extension commands.

4.3 _containers

Container classes and enums.

class repobee_plug._containers.**BaseParser**

Enumeration of base parsers that an extension command can request to have added to it.

BASE

Represents the base parser, which includes the `--user`, `--org-name`, `--base-url` and `--token` arguments.

STUDENTS

Represents the students parser, which includes the `--students` and *–students-file‘* arguments.

REPO_NAMES

Represents the repo names parser, which includes the `--master-repo-names` argument.

REPO_DISCOVERY

Represents the repo discovery parser, which adds both the `--master-repo-names` and the `--discover-repos` arguments.

MASTER_ORG

Represents the master organization parser, which includes the `--master-org` argument.

class repobee_plug._containers.**Deprecation**

Parameters

- **replacement** (*str*) – The functionality that replaces the deprecated functionality.
- **remove_by_version** (*str*) – A version number on the form `MAJOR.MINOR.PATCH` by which the deprecated functionality will be removed.

Create new instance of Deprecation(replacement, remove_by_version)

remove_by_version

Alias for field number 1

replacement

Alias for field number 0

class repobee_plug._containers.**ExtensionCommand**(*parser*, *name*, *help*, *description*, *callback*, *requires_api=False*, *requires_base_parsers=None*)

Class defining an extension command for the RepoBee CLI.

Parameters

- **parser** (`ExtensionParser`) – The parser to use for the CLI.
- **name** (`str`) – Name of the command.
- **help** (`str`) – Text that will be displayed when running `repobee -h`
- **description** (`str`) – Text that will be displayed when calling the `-h` option for this specific command. Should be elaborate in describing the usage of the command.
- **callback** (`Callable`[[`Namespace`, `Optional`[`API`]], `Optional`[`Mapping`[`str`, `Result`]]]) – A callback function that is called if this command is used on the CLI. It is

passed the parsed namespace and the platform API. It may optionally return a result mapping on the form (name: str -> List[Result]) that's reported by RepoBee.

- **requires_api** (bool) – If True, the base arguments required for the platform API are added as options to the extension command, and the platform API is then passed to the callback function. It is then important not to have clashing option names. If False, the base arguments are not added to the CLI, and None is passed in place of the API. If you include `REPO_DISCOVERY` in `requires_base_parsers`, then you *must* set this to True.
- **requires_base_parsers** (Optional[Iterable[BaseParser]]) – A list of `repobee_plug.BaseParser` that decide which base parsers are added to this command. For example, setting `requires_base_parsers = [BaseParser.STUDENTS]` adds the `--students` and `--students-file` options to this extension command's parser.

class repobee_plug._containers.**ExtensionParser**
An ArgumentParser specialized for RepoBee extension commands.

repobee_plug._containers.**HookResult**(*hook*, *status*, *msg*, *data=None*)
Backwards compat function.

Deprecated since version 0.12.0: Replaced by `Result`.

Return type Result

class repobee_plug._containers.**Result**(*name*, *status*, *msg*, *data=None*)
Container for storing results from hooks.

Parameters

- **name** (str) – Name to associate with this result. This is typically the name of the plugin that returns this result.
- **status** (Status) – Status of the plugin execution.
- **msg** (str) – A free-form result message.
- **data** (Optional[Mapping[Any, Any]]) – Semi-structured data in the form of a dictionary. All of the contents of the dictionary should be serializable as this is primarily used for JSON storage.

class repobee_plug._containers.**Review**

Parameters

- **repo** (Repo) – The reviewed repository.
- **done** (*bool*) – Whether or not the review is done.

Create new instance of Review(repo, done)

done
Alias for field number 1

repo
Alias for field number 0

class repobee_plug._containers.**ReviewAllocation**

Parameters

- **review_team** (Team) – The team of reviewers.
- **reviewed_team** (Team) – The team that is to be reviewed.

Create new instance of ReviewAllocation(review_team, reviewed_team)

review_team
Alias for field number 0

reviewed_team
Alias for field number 1

class repobee_plug._containers.**Status**
Status codes enums for Results.

SUCCESS
Signifies a plugin execution without any complications.

WARNING
Signifies a plugin execution with non-critical failures.

ERROR
Signifies a critical error during execution.

4.4 _corehooks

Hookspecs for repobee core hooks.

Core hooks provide the basic functionality of repobee. These hooks all have default implementations, but are overridden by any other implementation. All hooks in this module should have the *firstresult=True* option to the hookspec to allow for this dynamic override.

class repobee_plug._corehooks.**APIHook**
Hooks related to platform APIs.

api_init_requires()
Return which of the arguments to apimeta.APISpec.__init__ that the given API requires. For example, the GitHubAPI requires all, but the GitLabAPI does not require `user`.

Return type Tuple[str]

Returns Names of the required arguments.

get_api_class()
Return an API platform class. Must be a subclass of apimeta.API.

Returns An apimeta.API subclass.

class repobee_plug._corehooks.**PeerReviewHook**
Hook functions related to allocating peer reviews.

generate_review_allocations(*teams*, *num_reviews*)
Generate `ReviewAllocation` tuples from the provided teams, given that this concerns reviews for a single master repo.

The provided teams of students should be treated as units. That is to say, if there are multiple members in a team, they should always be assigned to the same review team. The best way to merge two teams `team_a` and `team_b` into one review team is to simply do:

```
team_c = apimeta.Team(members=team_a.members + team_b.members)
```

This can be scaled to however many teams you would like to merge. As a practical example, if teams `team_a` and `team_b` are to review `team_c`, then the following `ReviewAllocation` tuple, here called `allocation`, should be contained in the returned list.

```
review_team = apimeta.Team(members=team_a.members + team_b.members)
allocation = containers.ReviewAllocation(
    review_team=review_team,
    reviewed_team=team_c,
)
```

Note: Respecting the `num_reviews` argument is optional: only do it if it makes sense. It's good practice to issue a warning if num_reviews is ignored, however.

Parameters

- **team** – A list of `Team` tuples.
- **num_reviews** (`int`) – Amount of reviews each student should perform (and consequently amount of reviewers per repo)

Return type `List[ReviewAllocation]`

Returns

A list of `ReviewAllocation` tuples.

4.5 _exthooks

Hookspecs for repobee extension hooks.

Extension hooks add something to the functionality of repobee, but are not necessary for its operation. Currently, all extension hooks are related to cloning repos.

class repobee_plug._exthooks.**CloneHook**

Hook functions related to cloning repos.

act_on_cloned_repo(*path*, *api*)

Do something with a cloned repo.

Deprecated since version 0.12.0: This hook is has been replaced by `TaskHooks.clone_task()`. Once all known, existing plugins have been migrated to the new hook, this hook will be removed.

Parameters

- **path** (`Union[str, Path]`) – Path to the repo.
- **api** (`API`) – An instance of `repobee.github_api.GitHubAPI`.

Return type `Optional[Result]`

Returns optionally returns a Result namedtuple for reporting the outcome of the hook. May also return None, in which case no reporting will be performed for the hook.

clone_parser_hook(*clone_parser*)

Do something with the clone repos subparser before it is used used to parse CLI options. The typical task is to add options to it.

Deprecated since version 0.12.0: This hook is has been replaced by `TaskHooks.clone_task()`. Once all known, existing plugins have been migrated to the new hook, this hook will be removed.

Parameters **clone_parser** (`ArgumentParser`) – The `clone` subparser.

Return type `None`

config_hook(*config_parser*)
Hook into the config file parsing.

Parameters **config** – the config parser after config has been read.

Return type None

parse_args(*args*)
Get the raw args from the parser. Only called for the clone parser. The typical task is to fetch any values from options added in clone_parser_hook().

Parameters **args** (Namespace) – The full namespace returned by argparse.ArgumentParser.parse_args()

Return type None

class repobee_plug._exthooks.**ExtensionCommandHook**
Hooks related to extension commands.

create_extension_command()
Create an extension command to add to the RepoBee CLI. The command will be added as one of the top-level subcommands of RepoBee. This hook is called precisely once, and should return an *ExtensionCommand*.

```
def command(args: argparse.Namespace, api: apimeta.API)
```

The command function will be called if the extension command is used on the command line.

Note that the RepoBeeExtensionParser class is just a thin wrapper around argparse.ArgumentParser, and can be used in an identical manner. The following is an example definition of this hook that adds a subcommand called example-command, that can be called with repobee example-command.

```
import repobee_plug as plug

def callback(args: argparse.Namespace, api: plug.API) -> None:
    LOGGER.info("callback called with: {}, {}".format(args, api))

@plug.repobee_hook
def create_extension_command():
    parser = plug.RepoBeeExtensionParser()
    parser.add_argument("-b", "--bb", help="A useless argument")
    return plug.ExtensionCommand(
        parser=parser,
        name="example-command",
        help="An example command",
        description="Description of an example command",
        callback=callback,
    )
```

Important: If you need to use the api, you set requires_api=True in the ExtensionCommand. This will automatically add the options that the API requires to the CLI options of the subcommand, and initialize the api and pass it in.

See the documentation for *ExtensionCommand* for more details on it.

Return type ExtensionCommand

Returns A ExtensionCommand.

class repobee_plug._exthooks.**TaskHooks**
Hook functions relating to RepoBee tasks.

clone_task()
Create a task to run on a copy of a cloned student repo. This hook replaces the old `act_on_cloned_repo` hook.

Implementations of this hook should return a `Task`, which defines a callback that is called after all student repos have been cloned. See the definition of `Task` for details.

Return type `Task`

Returns A `Task` instance defining a RepoBee task.

setup_task()
Create a task to run on a copy of the master repo before it is pushed out to student repositories. This can for example be pre-flight checks of the master repo, or something else entirely.

Implementations of this hook should return a `Task`, which defines a callback that is called after the master repo has been safely copied, but before that copy is pushed out to student repositories. Note that any changes to the repository must be committed to actually show up in the student repositories.

Note: Structural changes to the master repo are not currently supported. Changes to the repository during the callback will not be reflected in the generated repositories. Support for preprocessing (such that changes do take effect) is a potential future feature.

Return type `Task`

4.6 _exceptions

Exceptions for repobee_plug.

exception repobee_plug._exceptions.**APIImplementationError**(**args*, ***kwargs*)
Raise when an API is defined incorrectly.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to `Exception`. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument `a` caused the error, then you should pass `a=a` as a kwarg so it can be introspected at a later time.

exception repobee_plug._exceptions.**ExtensionCommandError**(**args*, ***kwargs*)
Raise when an :py:class:~repobee_plug.containers.ExtensionCommand: is incorrectly defined.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to `Exception`. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument `a` caused the error, then you should pass `a=a` as a kwarg so it can be introspected at a later time.

exception repobee_plug._exceptions.**HookNameError**(**args*, ***kwargs*)
Raise when a public method in a class that inherits from *Plugin* does not have a hook name.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to Exception. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument a caused the error, then you should pass a=a as a kwarg so it can be introspected at a later time.

exception repobee_plug._exceptions.**PlugError**(**args*, ***kwargs*)
Base class for all repobee_plug exceptions.

Instantiate a PlugError.

Parameters

- **args** – List of positionals. These are passed directly to Exception. Typically, you should only pass an error message here.
- **kwargs** – Keyword arguments to indicate what went wrong. For example, if the argument a caused the error, then you should pass a=a as a kwarg so it can be introspected at a later time.

4.7 _name

Utility functions relating to RepoBee's naming conventions.

repobee_plug._name.**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

repobee_plug._name.**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.

repobee_plug._name.**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.

4.8 _serialize

JSON serialization/deserialization functions.

repobee_plug._serialize.**json_to_result_mapping**(*json_string*)
Deserialize a JSON string to a mapping `repo_name: str -> hook_results: List[Result]`

Return type Mapping[str, List[Result]]

repobee_plug._serialize.**result_mapping_to_json**(*result_mapping*)
Serialize a result mapping `repo_name: str -> hook_results: List[Result]` to JSON.

Return type str

4.9 _tasks

Task data structure and related functionality.

class repobee_plug._tasks.**Task**(*act*, *add_option=None*, *handle_args=None*, *persist_changes=False*)
A data structure for describing a task. Tasks are operations that plugins can define to run on for example cloned student repos (a clone task) or on master repos before setting up student repos (a setup task). Only the `act` attribute is required, all other attributes can be omitted.

The callback methods should have the following headers.

```
def act(
    path: pathlib.Path, api: repobee_plug.API
) -> Optional[containers.Result]:

def add_option(parser: argparse.ArgumentParser) -> None:

def handle_args(args: argparse.Namespace) -> None:
```

Note: The functions are called in the following order: `add_option` -> `handle_args` -> `act`.

Important: The `act` callback should *never* change the Git repository it acts upon (e.g. running commands such as `git add`, `git checkout` or `git commit`). This can have adverse and unexpected effects on RepoBee's functionality. It is however absolutely fine to change the files in the Git working tree, as long as nothing is added or committed.

Each callback is called at most once. They are not guaranteed to execute, because there may be an unexpected crash somewhere else, or the plugin may not come into scope (for example, a clone task plugin will not come into scope if `repobee setup` is run). The callbacks can do whatever is appropriate for the plugin, except for changing any Git repositories. For information on the types used in the callbacks, see the Python stdlib documentation for `argparse`.

As an example, a simple clone task can be defined like so:

```
import repobee_plug as plug

def act(path, api):
    return plug.Result(
        name="example",
        msg="IT LIVES!",
        status=plug.Status.SUCCESS
    )

@plug.repobee_hook
def clone_task():
    return plug.Task(act=act)
```

If your task plugin also needs to access the configuration file, then implement the separate `config_hook` hook. For more elaborate instructions on creating tasks, see the tutorial.

Parameters

- **act** (Callable[[Path, API], Result]) – A required callback function that takes the path to a repository worktree and an API instance, and optionally returns a Result to report results.
- **add_option** (Optional[Callable[[ArgumentParser], None]]) – An optional callback function that adds options to the CLI parser.
- **handle_args** (Optional[Callable[[Namespace], None]]) – An optional callback function that receives the parsed CLI args.
- **persist_changes** (bool) – If True, the task requires that changes to the repository that has been acted upon be persisted. This means different things in different contexts (e.g. whether the task is executed in a clone context or in a setup context), and may not be supported for all contexts.

4.10 _deprecation

Module with functions for dealing with deprecation.

repobee_plug._deprecation.**deprecate**(*remove_by_version*, *replacement=None*)

Return a function that can be used to deprecate functions. Currently this is only used for deprecation of hook functions, but it may be expanded to deprecated other things in the future.

Parameters

- **remove_by_version** (str) – A string that should contain a version number.
- **replacement** (Optional[str]) – An optional string with the name of the replacing function.

Return type Callable[[~T], ~T]

Returns A function

repobee_plug._deprecation.**deprecated_hooks**()

Return type Mapping[str, Deprecation]

Returns A mapping of hook names to Deprecation tuples.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

M

N

O

P

R

S

T

V

W