Command Line Achievements Documentation

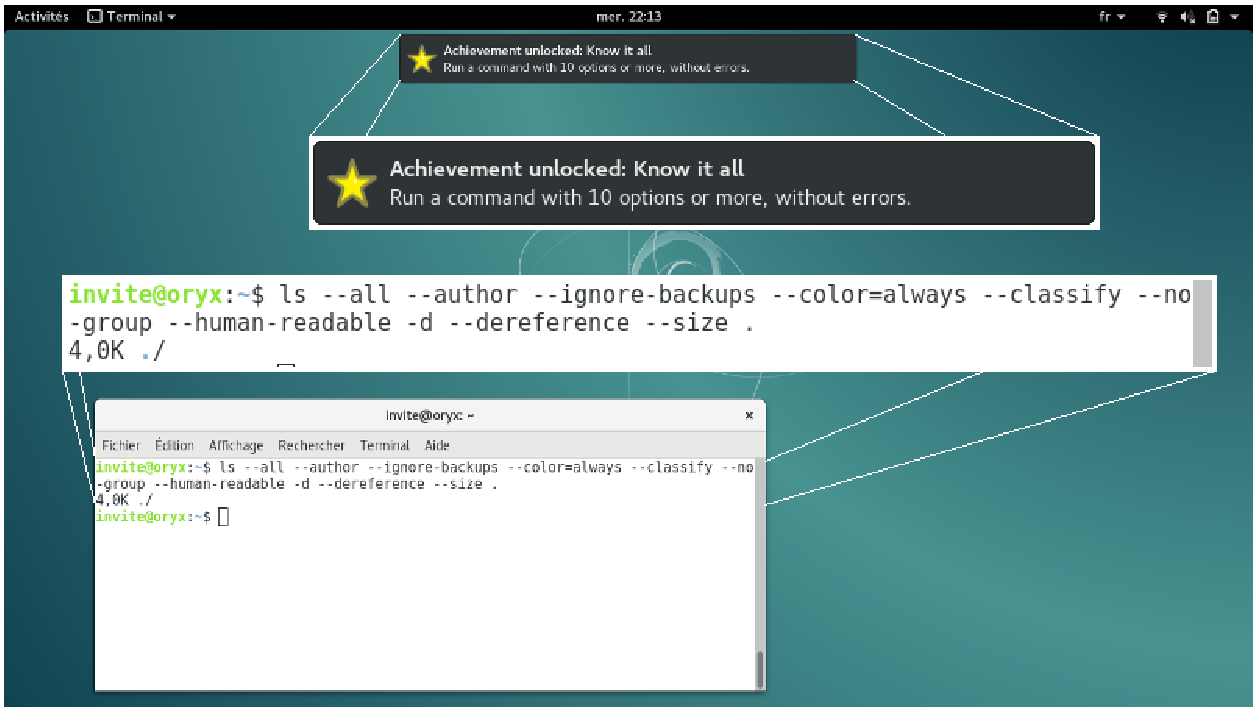
Release 0.1.0

Louis Paternault

Dec 25, 2022

CONTENTS

Command line was all fun when it was invented fifty years ago, but it can no longer compete with modern interfaces, glowing windows, shiny buttons, and so on… Command Line Achievements (later abbreviated CLAchievements or CLA) aims to solve this, by making command line fun again! Inspired by video game achievements, it unlocks achievements when users meets the fulfillement conditions.



Achievement unlocked: Know it all
Run a command with 10 options or more, without errors.

```
invite@oryx:~$ ls --all --author --ignore-backups --color=always --classify --no
-group --human-readable -d --dereference --size .
4,0K ./
invite@oryx:~$
```

Warning: CLAchievements is not mature yet. Let's say it is a proof of concept (a silly concept, but still a concept).

Before using this software, please read the *warning*.

Contents:

CHAPTER

ONE

INSTALL AND ENABLE

Warning: *Installing* CLAchievements is not enough: it has to be *enabled*.

1.1 Install

1.1.1 PyGObject

CLAchievements uses PyGObject to display the achievements (other methods might be supported later). Thus, it must be installed, either system-wide (if CLAchievements is not installed in a virtualenv, or if the virtualenv has been created with option `--system-site-packages`), or as a dependency (see the *extra* `pgi` dependency below).

1.1.2 From sources

- Download: https://pypi.python.org/pypi/clachievements
- Install (in a *virtualenv*, if you do not want to mess with your distribution installation system):

  ```
  python3 setup.py install
  ```

  Or, to install the `pgi` dependency as well:

  ```
  python3 setup.py install[pgi]
  ```

1.1.3 From pip

Use:

```
pip install clachievements
```

Or, if you need the `pgi` dependency as well:

```
pip install clachievements[pgi]
```

1.1.4 Quick and dirty Debian (and Ubuntu?) package

This requires stdeb to be installed:

```
python3 setup.py --command-packages=stdeb.command bdist_deb
sudo dpkg -i deb_dist/clachievements-<VERSION>_all.deb
```

The `PyGObject` dependency is proposed as an optional requirement.

1.2 Enable

Once CLAchievements is installed, it does not work yet. Running `ls` will not trigger any achievement: you will to wrap it using CLAchievements by running `clachievements run ls`.

Replacing `ls` by `clachievements run ls` will change your habits. You do not want it. So, it should be aliased: `alias ls="clachievements run ls"`.

All the commands triggering achievements should be aliased. To ease this, the clachievements command provides a sub-command `clachievements alias`, which display the shell code generating all the required aliases. Thus, in your `.bashrc` (or `.watheverrc`), write the line `$(clachievements alias)` to enable every aliases.

1.3 Check

To check if CLAchievements is enabled, run `ls` in a terminal. If you see the `So it begins...` achievement unlocked, it works. Otherwise, it does not… yet.

If you are not sure about wether CLAchievements works or not, reset the achievements using `clachievements reset`, and run `ls` again to test it.

CHAPTER TWO

USAGE

Here are the command line options for *clachievements*.

Various tools for Command Line Achievements.

```
usage: clachievements [-h] [--version] {alias,report,reset,run} ...
```

2.1 Named Arguments

--version	Show version

2.2 Subcommands

List of available subcommands.

subcommand	Possible choices: alias, report, reset, run

2.3 Sub-commands

2.3.1 alias

Display code to create aliases.

```
clachievements alias [-h]
```

2.3.2 report

Display a progress summary.

```
clachievements report [-h]
```

2.3.3 reset

Reset progress.

```
clachievements reset [-h]
```

2.3.4 run

Run binaries, unlocking achievements if relevant.

```
clachievements run [-h]
```

CHAPTER

THREE

WRITE YOUR OWN ACHIEVEMENT

3.1 Achievement without persistent data

Suppose you want to create an achievement `Foo` awarded when user successfully run a command on a file `foo`. Let's write this achievement.

3.1.1 Meta-information

First, we need to define a class and define meta-information: any achievement is a subclass of *`Achievement`*. Two arguments are compulsory:

- *`title`*: if `None`, your class is an abstract achievement, meant to be subclassed; if a string, your achievement is an *actual* achievement. See the *`class documentation`* for other attributes;
- *description*: your achievement must have a description. The first non-empty line of your class docstring is used, unless `_description` is defined, when it is used instead.

See *`the class documentation`* to get more information about other attributes.

```
from clachievements.achievements import Achievement
from clachievements.testutils import test_lock, test_unlock

class Foo(Achievement):
    """Successfully run a command on file `foo`."""

    title = "Foo"
```

3.1.2 Unlocking the achievement

Great: you have an achievement. But it is never unlocked: it will be frustrating for the user.

An achievement is a context manager: its `__enter__()` and `__exit__()` methods are called before and after the actual system call. They can be used to test the command line, the environment before and after the command, etc.

Here, we test that:

- `foo` is a positional argument;
- the command did not fail.

If so, we call *`unlock()`* to unlock the argument. It ensures that the argument is marked as unlocked, and it displays a pop-up to notify the user. No need to make sure that parallel calls to your achievement might unlock it at the same time: it is handled within the *`unlock()`* method itself.

```python
from clachievements.achievements import Achievement
from clachievements.testutils import test_lock, test_unlock

class Foo(Achievement):
    """Successfully run a command on file `foo`."""

    title = "Foo"

    def __exit__(self, exc_type, exc_value, traceback):
        if "foo" in self.command.positional:
            if isinstance(exc_value, SystemExit):
                if exc_value.code == 0:
                    self.unlock()
```

3.1.3 Testing

If we are done, the achievement will work, but the unit tests will fail. An achievement *must* define a test that unlock the achievement.

Each achievement must define a static or class method, **decorated** with `test_unlock()`. This method must iterate strings which are shell commands, unlocking the achievement. To be wrapped by CLAchievements, system calls must use string substitution: `"foo bar"` will call the `foo` binary, *not wrapped* by CLAchievements, where `"{bin.foo} bar"` will call the `foo` binary, wrapped by CLAchievements.

You can add as many test methods as you want. You can also define test methods that must not unlock achievements, by decorating them with `test_lock()`.

When performing tests, each test method is run inside an empty temporary directory, which will be deleted afterward.

```python
from clachievements.achievements import Achievement
from clachievements.testutils import test_lock, test_unlock

class Foo(Achievement):
    """Successfully run a command on file `foo`."""

    title = "Foo"

    def __exit__(self, exc_type, exc_value, traceback):
        if "foo" in self.command.positional:
            if isinstance(exc_value, SystemExit):
                if exc_value.code == 0:
                    self.unlock()

    @staticmethod
    @test_unlock
    def test_touch():
        yield "{bin.touch} foo"

    @staticmethod
    @test_lock
    def test_ls():
      yield "{bin.ls} foo"
```

3.2 Achievement with persistent data

Now, we want a new achievement `FooBar` to be triggered when 50 successful commands have been run on a file `foo`. Let's do this.

To do this, we have to store the number of successful commands. A class is defined to ease this process: *`SimplePersistentDataAchievement`*. It is wrong (see below), but is works for simple cases.

When using this class, a row is created in the CLAchievements database with this achievement name.

- The first time this achievement is created, this row is filled with the content of attribute *`default_data`*.
- When accessing to *`data`*, data is read from the database.
- When assigning a value to *`data`*, data is written to the database.

Any `picklable` data can be stored using this method.

This is simple, but this is not robust to concurrent access: if an integrity error occurs when assigning a value to *`data`*, it is silently ignored.

With this example achievement, if I run this argument 50 times in parallel, about 30 of the assignments are ignored. If I were to design a life critical application, this would be a big issues. But this is only a game: it does not work perfectly, but it is so much simpler to implement!

```
from clachievements.achievements import SimplePersistentDataAchievement
from clachievements.testutils import test_lock, test_unlock

class FooBar(SimplePersistentDataAchievement):
    """Successfully run 50 command on file `foo`."""

    title = "FooBar"
    default_data = 0

    def __exit__(self, exc_type, exc_value, traceback):
        if "foo" in self.command.positional:
            if isinstance(exc_value, SystemExit):
                if exc_value.code == 0:
                    self.data += 1
        if self.data >= 50:
            self.unlock()

    @staticmethod
    @test_lock
    def test_touch():
        for _ in range(49):
            yield "{bin.touch} foo"

    @staticmethod
    @test_unlock
    def test_ls_touch():
        for _ in range(25):
            yield "{bin.touch} foo"
            yield "{bin.ls} foo"
```

3.3 More

Suppose this error-prone persistent data management does not suit you. Just write your own: within the achievement, the `sqlite3 database connection` is available as `self.database.conn`. Do whatever you want with it (without breaking other plugin databases)!

In this case, to be sure not to mess with tables of CLA core or other plugins, use the tables named (case insensitive) `achievement_YourPluginName` or `achievement_YourPluginName_*`.

Methods *`first()`* and *`last()`* can be used to initialize or clean the achievement: the first one is called the first time the achievement is ever loaded (so it can be used to create some tables into the database), while the last one is called when the achievement has just been unlocked (so it can be used to clean stuff). Both these methods are meant to be subclassed, and are expected to call `super().first(...)` at the beginning of their code.

CHAPTER

FOUR

MODULES

4.1 Command

class Command(*argv*)

Parse command line call, for easy access to parameters.

> **Warning:** This class does not work, on purpose. Correctly parsing command line depends on each command, and implementing it correctly would mean re-implementing the parsing process used by every binary that is to be wrapped with CLAchievements. This is not going to happen. What is done is:
>
> - Any argument *not* starting with `-` is a positional argument.
> - Any argument starting with a single `-` is a list of short options (`-foo` is equivalent to `-f -o -o`). Those options *do not have* any arguments.
> - Any argument starting with double `--` is a long option. If it contains a `=`, it is intepreted as an option with its argument; otherwise, it does not have any arguments.

The available attributes are:

bin

Base name of the wrapped binary (more or less equivalent to `os.path.basename(sys.argv[0])`).

short

`multidict.MultiDict` of short command line arguments (that is, arguments starting with a single `-`). Keys are the arguments, and values are the options to the arguments. See the warning at the beginning of the documentation of this class.

long

`multidict.MultiDict` of long command line arguments (that is, arguments starting with a double `-`). Keys are the arguments, and values are the options to the arguments. See the warning at the beginning of the documentation of this class.

positional

List of positional arguments (that is, arguments not starting with -).

argv

Complete list of arguments (as one would expect from `sys.argv`).

The following doctest serves as an example.

```
>>> command = Command("/usr/bin/foo tagada -bar --baz --baz=plop tsoin tsoin".
↪split())
>>> command.bin
'foo'
>>> command.short
<MultiDict('b': None, 'a': None, 'r': None)>
>>> command.long
<MultiDict('baz': None, 'baz': 'plop')>
>>> command.argv
['/usr/bin/foo', 'tagada', '-bar', '--baz', '--baz=plop', 'tsoin', 'tsoin']
>>> command.positional
['tagada', 'tsoin', 'tsoin']
```

4.2 Achievements

class Achievement(*command*, *database*)

Achievement: Something that is unlocked when user perform the right commands.

A how-to is available in the *Write your own achievement* section, which illustrates this class documentation.

This class is a context manager. The `__enter__()` method is called before the actual wrapped command call, and the `__exit__()` method is called after it. One of those method must call *`unlock()`* when the conditions to fulfill the achievement are met.

_description = None

Description of the achievement. If *None*, the first non-empty line of the class docstring is used instead.

bin = None

List of binaries loading this achievement. If *None*, this achievement is always loaded.

first()

This method is called once: when this achievement is loaded for the first time.

This method is meant to be subclassed.

icon = 'star.svg'

File name of the icon (relative to the data directory).

last()

This method is called once: when this achievement has just been unlocked.

This method is meant to be subclassed.

title = None

Title of the achievement. If *None*, the class is an abstract achievement, to be subclassed.

`unlock()`

Called when achievement is unlocked.

- Mark this achievement as unlocked in the database.
- Notify user.

This method is to be called by one of the `__enter__()` or `__exit__()` method when the conditions to unlock the achievement are fulfilled.

`class SimplePersistentDataAchievement`(*command*, *database*)

Achievement, with a simple way to store data into a database.

It is very simple to use, since accessing or writing to `self.data` will automatically read or write data from the database.

But the cost is that concurrent access to the database *will* lead to errors. For instance, on a test, running fifty concurrent calls to `self.data += 1` only incremented `self.data` by about twenty values.

This is wrong but:

- this is just a game, so there is no important consequence to this error;
- this is a very simple class. If you want a more robust one, please provide a patch.

`property data`

`Picklable` persistent data, specific to this achievement.

Note: Database is not locked when reading or writing this data. That is, concurrent runs of `self.data += 1` are not guaranteed to succeed.

Note: Be careful to call `self.data = MY_NEW_DATA` to store your updated data. This means that, if `self.data` is a dictionary, `self.data.update({"foo": "bar"})` will not store anything.

`default_data = None`

Data stored as this achievement persistent data when this achievement is met for the first time.

4.3 Test utils

`test_lock`(*func*)

Decorator for test methods keeping the achievement locked.

To be applied to methods of *`Achievement`*.

Those methods must iterate over shell commands (as strings). Executing those commands must not unlock the achievement. Otherwise, the corresponding test will fail.

`test_unlock`(*func*)

Decorator for test methods unlocking the achievement.

To be applied to methods of *`Achievement`*.

Those methods must iterate over shell commands (as strings). Executing those commands must unlock the achievement. Otherwise, the corresponding test will fail.

CHAPTER

FIVE

DOES IT WORK?

I would say CLAchievements works if user cannot distinguish wrapped commands from original commands, excepted the *Achievement unlocked* text popping up from times to times.

5.1 What works

When the command ends on its own, the standard input, and standard and error output are transmitted to the wrapped command, and the return code of the command is the expected one (the one of the wrapped command).

5.2 What does not work

- Interruptions: When a command is interrupted (by using Ctrl-C or kill), user sees the internals of CLAchievements.

```
$ cat
^CTraceback (most recent call last):
  File "/home/louis/.virtualenvs/clachievements/bin/clachievements", line 9, in
↪<module>
    load_entry_point('clachievements==0.1.0', 'console_scripts', 'clachievements')()
  (...)
  File "/usr/lib/python3.5/subprocess.py", line 1608, in _try_wait
    (pid, sts) = os.waitpid(self.pid, wait_flags)
KeyboardInterrupt
$
```

  It should act as if the wrapped command had been interrupted.

- It is incredibly slow: for instance, running one thousand *ls* is about 600 times slower using CLAchievements than using the original *ls*.

```
$ time bash -c 'for i in $(seq 1000); do clachievements run ls > /dev/null; done'
real    7m57.569s
user    6m3.960s
sys     0m21.536s
$ time bash -c 'for i in $(seq 1000); do ls > /dev/null; done'
real    0m0.790s
user    0m0.024s
sys     0m0.080s
```

  This is a real problem, and addressing it might mean rewriting everything from scratch…

CHAPTER SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX