Evennia Documentation

Release 0.6

The Evennia community

Jul 16, 2017

Contents

1 Evennia Introduction **3**
- 1.1 Can I test it somewhere? . . . 4
- 1.2 Brief summary of features . . . 4
- 1.3 What you need to know to work with Evennia . . . 5

2 How To Get And Give Help **7**
- 2.1 How to *get* Help . . . 7
- 2.2 How to *give* Help . . . 7

3 Contributing **9**
- 3.1 Spreading the word . . . 9
- 3.2 Donations . . . 9
- 3.3 Help with Documentation . . . 9
- 3.4 Contributing through a forked repository . . . 10
- 3.5 Contributing with Patches . . . 10
- 3.6 Contributing with Contribs . . . 10

4 Soft Code **13**
- 4.1 Examples of Softcode . . . 13
- 4.2 Problems with Softcode . . . 14
- 4.3 Changing Times . . . 15
- 4.4 Our Solution . . . 15
- 4.5 Your Solution . . . 16

5 Using MUX as a Standard **17**
- 5.1 Documentation policy . . . 17

6 Game Planning **19**
- 6.1 Planning (step 1) . . . 19
- 6.2 Coding (step 2) . . . 21
- 6.3 World Building (step 3) . . . 22
- 6.4 Alpha Release . . . 22
- 6.5 Beta Release/Perpetual Beta . . . 23
- 6.6 Congratulate yourself! . . . 23

7 Installation and setup **25**
- 7.1 Getting Started . . . 25

7.2 Start Stop Reload . . . 31
7.3 Choosing An SQL Server . . . 35
7.4 Inspecting database data . . . 36
7.5 Apache Config . . . 37
7.6 Updating Your Game . . . 38
7.7 Online Setup . . . 40

8 Admin Documentation **47**
8.1 Banning . . . 47
8.2 IRC . . . 49
8.3 RSS . . . 50
8.4 Text Encodings . . . 51
8.5 Internationalization . . . 52

9 Builder Documentation **55**
9.1 Building Quickstart . . . 55
9.2 Connection Screen . . . 59
9.3 Batch Processors . . . 60
9.4 Batch Command Processor . . . 61
9.5 Batch Code Processor . . . 64
9.6 Building Permissions . . . 68

10 Developer Basics **71**
10.1 Coding Introduction . . . 71
10.2 Version Control . . . 73
10.3 Quirks . . . 84
10.4 Licensing . . . 86

11 Server Components **87**
11.1 Directory Overview . . . 87
11.2 Server Conf . . . 88
11.3 Portal And Server . . . 90
11.4 Sessions . . . 90
11.5 Commands . . . 93
11.6 Command Sets . . . 100
11.7 Typeclasses . . . 107
11.8 Objects . . . 112
11.9 Scripts . . . 115
11.10 Players . . . 119
11.11 Communications . . . 121
11.12 Attributes . . . 123
11.13 Locks . . . 129
11.14 Permissions . . . 132
11.15 Help System . . . 136
11.16 Nicks . . . 138
11.17 Advanced note . . . 140
11.18 Tags . . . 140
11.19 Web Features . . . 142
11.20 TickerHandler . . . 145
11.21 Spawner . . . 147

12 Coding utilities **151**
12.1 New Models . . . 151
12.2 Execute Python Code . . . 155
12.3 Profiling . . . 157

12.4 Unit Testing . . . 159
12.5 Coding Utils . . . 165
12.6 Async Process . . . 169

13 Tutorials **175**
13.1 First Steps Coding . . . 175
13.2 Tutorial for basic MUSH like game . . . 179
13.3 Adding Command Tutorial . . . 190
13.4 Adding Object Typeclass Tutorial . . . 193
13.5 Tutorial World Introduction . . . 195
13.6 Command Duration . . . 197
13.7 Command Prompt . . . 201
13.8 Default Exit Errors . . . 204
13.9 Implementing a game rule system . . . 206
13.10 Weather Tutorial . . . 210
13.11 Zones . . . 211
13.12 Web Tutorial . . . 212
13.13 Web Character View Tutorial . . . 214

14 Appendix **219**
14.1 Links . . . 219
14.2 Default Command Help . . . 222
14.3 Evennia Devel . . . 271

This is the manual of Evennia, the open source Python MU* creation system. You should hopefully find all you need to know about coding with, extending and using the code base among these pages. If you have further questions you are welcome to ask them in the Developer online chat or, if you don't have time to hang around for an answer, on the Mailing list.

Please note that this ReadTheDocs version of the documentation is auto-converted from the original documentation found in the Evennia wiki. It represents the wiki as it looked *on Sunday May 28, at 21:02 (GMT+1)*.

The converted version is supplied as a service for those wanting to have access to the documentation in an offline format, on their tablet or printed on paper. The conversion unfortunately means that there are occational conversion artifacts. Notably inter-documentation links and images may not work. This should hopefully not affect readability or understanding, but is something to keep in mind.

CHAPTER 1

Evennia Introduction

> *A MUD (originally Multi-User Dungeon, with later variants Multi-User Dimension and Multi-User Domain) is a multiplayer real-time virtual world described primarily in text. MUDs combine elements of role-playing games, hack and slash, player versus player, interactive fiction and online chat. Players can read or view descriptions of rooms, objects, other players, non-player characters, and actions performed in the virtual world. Players typically interact with each other and the world by typing commands that resemble a natural language.* - Wikipedia

If you are reading this, it's quite likely you are dreaming of creating and running a text-based massively-multiplayer game (MUD/MUX/MUSH etc) of your very own. You might just be starting to think about it, or you might have lugged around that *perfect* game in your mind for years ... you know *just* how good it would be, if you could only make it come to reality. We know how you feel. That is, after all, why Evennia came to be.

Evennia is in principle a MUD-building system: a bare-bones Python codebase and server intended to be highly extendable for any style of game. "Bare-bones" in this context means that we try to impose as few game-specific things on you as possible. So whereas we for convenience offer basic building blocks like objects, characters, rooms, default commands for building and administration etc, we don't prescribe any combat rules, mob AI, races, skills, character classes or other things that will be different from game to game anyway. It is possible that we will offer some such systems as contributions in the future, but these will in that case all be optional.

What we *do* however, is to provide a solid foundation for all the boring database, networking, and behind-the-scenes administration stuff that all online games need whether they like it or not. Evennia is *fully persistent*, that means things you drop on the ground somewhere will still be there a dozen server reboots later. Through Django we support a large variety of different database systems (a database is created for you automatically if you use the defaults).

Using the full power of Python throughout the server offers some distinct advantages. All your coding, from object definitions and custom commands to AI scripts and economic systems is done in normal Python modules rather than some ad-hoc scripting language. The fact that you script the game in the same high-level language that you code it in allows for very powerful and custom game implementations indeed.

The server ships with a default set of player commands that are similar to the MUX command set. We *do not* aim specifically to be a MUX server, but we had to pick some default to go with (see this for more about our original motivations). It's easy to remove or add commands, or to have the command syntax mimic other systems, like Diku, LP, MOO and so on. Or why not create a new and better command system of your own design.

Can I test it somewhere?

Evennia's demo server can be found at demo.evennia.com. If you prefer to connect to the demo via your own telnet client you can do so at `silvren.com`, port `4280`. Here is a screenshot.

Once you installed Evennia yourself it comes with its own tutorial - this shows off some of the possibilities *and* gives you a small single-player quest to play. The tutorial takes only one single in-game command to install as explained here.

Brief summary of features

Technical

- Game development is done by the server importing your normal Python modules. Specific server features are implemented by overloading hooks that the engine calls appropriately.
- All game entities are simply Python classes that handle database negotiations behind the scenes without you needing to worry.
- Command sets are stored on individual objects (including characters) to offer unique functionality and object-specific commands. Sets can be updated and modified on the fly to expand/limit player input options during play.
- Scripts are used to offer asynchronous/timed execution abilities. Scripts can also be persistent. There are easy mechanisms to thread particularly long-running processes and built-in ways to start "tickers" for games that wants them.
- In-game communication channels are modular and can be modified to any functionality, including mailing systems and full logging of all messages.
- Server can be fully rebooted/reloaded without users disconnecting.
- A Player can freely connect/disconnect from game-objects, offering an easy way to implement multi-character systems and puppeting.
- Each Player can optionally control multiple Characters/Objects at the same time using the same login information.
- Spawning of individual objects via a prototypes-like system.
- Tagging can be used to implement zones and object groupings.
- All source code is extensively documented.
- Unit-testing suite, including tests of default commands and plugins.

Default content

- Basic classes for Objects, Characters, Rooms and Exits
- Basic login system, using the Player's login name as their in-game Character's name for simplicity
- "MUX-like" command set with administration, building, puppeting, channels and social commands
- In-game Tutorial
- Contributions folder with working, but optional, code such as alternative login, menus, character generation and more

Standards/Protocols supported

- Telnet with mud-specific extensions (MCCP, MSSP, TTYPE, MSDP, GMCP, MXP links)
- ANSI, xterm256 colours
- SSH
- SSL
- TCP/websocket browser web client, with ajax/comet fallback for older browsers
- HTTP - Website served by in-built webserver and connected to same database as game.
- IRC - external IRC channels can be connected to in-game chat channels
- RSS feeds can be echoed to in-game channels (things like Twitter can easily be added)
- Several different databases supported (SQLite3, MySQL, PostgreSQL, . . .)

For more extensive feature information, see the Developer Central.

What you need to know to work with Evennia

Assuming you have Evennia working (see the quick start instructions) and have gotten as far as to start the server and connect to it with the client of your choice, here's what you need to know depending on your skills and needs.

I don't know (or don't want to do) any programming - I just want to run a game!

Evennia comes with a default set of commands for the Python newbies and for those who need to get a game running *now*. Stock Evennia is enough for running a simple 'Talker'-type game - you can build and describe rooms and basic objects, have chat channels, do emotes and other things suitable for a social or free-form MU*. Combat, mobs and other game elements are not included, so you'll have a very basic game indeed if you are not willing to do at least *some* coding.

I know basic Python, or I am willing to learn

Evennia's source code is extensively documented and is viewable online. We also have a comprehensive online manual with lots of examples. But while Python is considered a very easy programming language to get into, you do have a learning curve to climb if you are new to programming. You should probably sit down with a Python beginner's tutorial (there are plenty of them on the web if you look around) so you at least know what you are seeing. See also our link page for some reading suggestions. To efficiently code your dream game in Evennia you don't need to be a Python guru, but you do need to be able to read example code containing at least these basic Python features:

- Importing and using python modules
- Using variables, conditional statements, loops and functions
- Using lists, dictionaries and list comprehensions
- Doing string handling and formatting
- Have a basic understanding of object-oriented programming, using Classes, their methods and properties

Obviously, the more things you feel comfortable with, the easier time you'll have to find your way. With just basic knowledge you should be able to define your own Commands, create custom Objects as well as make your world come alive with basic Scripts. You can definitely build a whole advanced and customized game from extending Evennia's examples only.

I know my Python stuff and I am willing to use it!

Even if you started out as a Python beginner, you will likely get to this point after working on your game for a while. With more general knowledge in Python the full power of Evennia opens up for you. Apart from modifying commands, objects and scripts, you can develop everything from advanced mob AI and economic systems, through sophisticated combat and social mini games, to redefining how commands, players, rooms or channels themselves work. Since you code your game by importing normal Python modules, there are few limits to what you can accomplish.

If you *also* happen to know some web programming (HTML, CSS, Javascript) there is also a web presence (a website and a mud web client) to play around with ...

Where to from here?

From here you can continue browsing the online documentation to find more info about Evennia. Or you can jump into the Tutorials and get your hands dirty with code right away. If you want more reading there is also a longer article about Evennia in Volume 5, Issue 1 of the Imaginary Realities e-magazine.

Some more hints:

1. Get engaged in the community. Make an introductory post to our mailing list/forum and get to know people. It's also highly recommended you hop onto our Developer chat on IRC. This allows you to chat directly with other developers new and old as well as with the devs of Evennia itself. This chat is logged (you can find links on http://www.evennia.com) and can also be searched from the same place for discussion topics you are interested in.
2. Read the Game Planning wiki page. It gives some ideas for your work flow and the state of mind you should aim for - including cutting down the scope of your game for its first release.
3. Do the Tutorial for basic MUSH-like game carefully from beginning to end and try to understand what does what. Even if you are not interested in a MUSH for your own game, you will end up with a small (very small) game that you can build or learn from.

CHAPTER 2

How To Get And Give Help

How to *get* Help

If you cannot find what you are looking for in the online documentation, here's what to do:

- If you think the documentation is not clear enough and are short on time, fill in our quick little online form and let us know - no login required. Maybe the docs need to be improved or a new tutorial added! Note that while this form is useful as a suggestion box we cannot answer questions or reply to you. Use the discussion group or chat (linked below) if you want feedback.
- If you have trouble with a missing feature or a problem you think is a bug, go to the issue tracker and search to see if has been reported/suggested already. If you can't find an existing entry create a new one.
- If you need help, want to start a discussion or get some input on something you are working on, make a post to the discussions group This is technically a 'mailing list', but you don't need to use e-mail; you can post and read all messages just as easily from your browser via the online interface.
- If you want more direct discussions with developers and other users, consider dropping into our IRC chat channel #evennia on the *Freenode* network. Please note however that you have to be patient if you don't get any response immediately; we are all in very different time zones and many have busy personal lives. So you might have to hang around for a while - you'll get noticed eventually!

How to *give* Help

Evennia is a completely non-funded project. It relies on the time donated by its users and developers in order to progress.

The first and easiest way you as a user can help us out is by taking part in community discussions and by giving feedback on what is good or bad. Report bugs you find and features you lack to our issue tracker. Just the simple act of letting developers know you are out there using their program is worth a lot. Generally mentioning and reviewing Evennia elsewhere is also a nice way to spread the word.

If you'd like to help develop Evennia more hands-on, here are some ways to get going:

- Look through our online documentation wiki and see if you can help improve or expand the documentation (even small things like fixing typos!). You don't need any particular permissions to edit the wiki.
- Send a message to our discussion group and/or our IRC chat asking about what needs doing, along with what your interests and skills are.
- Take a look at our issue tracker and see if there's something you feel like taking on.
- Check out the Contributing page on how to practically contribute with code using github.

. . . And finally, if you want to help motivate and support development you *can* also drop some coins in the developer's cup. You can make a donation via PayPal or, even better, become an Evennia patron on Patreon! This is a great way to tip your hat and show that you appreciate the work done with the server!

CHAPTER 3

Contributing

Wanna help out? Great! Here's how.

Spreading the word

Even if you are not keen on working on the server code yourself, just spreading the word is a big help - it will help attract more people which leads to more feedback, motivation and interest. Consider writing about Evennia on your blog or in your favorite (relevant) forum. Write a review somewhere (good or bad, we like feedback either way). Rate it on places like ohloh. Talk about it to your friends . . . that kind of thing.

Donations

The best way to support Evennia is to become an Evennia patron. Evennia is a free, open-source project and any monetary donations you want to offer are completely voluntary. See it as a way of announcing that you appreciate the work done - a tip of the hat! A patron donates a (usually small) sum every month to show continued support. If this is not your thing you can also show your appreciation via a one-time donation (this is a PayPal link but you don't need PayPal yourself).

Help with Documentation

Evennia depends heavily on good documentation and we are always looking for extra eyes and hands to improve it. Even small things such as fixing typos are a great help!

The documentation is a wiki and as long as you have a GitHub account you can edit it. It can be a good idea to discuss in the chat or forums if you want to add new pages/tutorials. Otherwise, it goes a long way just pointing out wiki errors so we can fix them (in an Issue or just over chat/forum).

Contributing through a forked repository

We always need more eyes and hands on the code. Even if you don't feel confident with tackling a bug or feature, just correcting typos, adjusting formatting or simply *using* the thing and reporting when stuff doesn't make sense helps us a lot.

The most elegant way to contribute code to Evennia is to use GitHub to create a *fork* of the Evennia repository and make your changes to that. Refer to the Forking Evennia version control instructions for detailed instructions.

Once you have a fork set up, you can not only work on your own game in a separate branch, you can also commit your fixes to Evennia itself. Make separate branches for all Evennia additions you do - don't edit your local master branch directly. It will make your life a lot easier. If you have a change that you think is suitable for the main Evennia repository, you issue a Pull Request. This will let Evennia devs know you have stuff to share.

Contributing with Patches

To help with Evennia development it's recommended to do so using a fork repository as described above. But for small, well isolated fixes you are also welcome to submit your suggested Evennia fixes/addendums as a patch.

You can include your patch in an Issue or a Mailing list post. Please avoid pasting the full patch text directly in your post though, best is to use a site like Pastebin and just supply the link.

Contributing with Contribs

While Evennia's core is pretty much game-agnostic, it also has a `contrib/` directory. The `contrib` directory contains game systems that are specialized or useful only to certain types of games. Users are welcome to contribute to the `contrib/` directory. Such contributions should always happen via a Forked repository as described above.

- If you are unsure if your idea/code is suitable as a contrib, *ask the devs before putting any work into it*. This can also be a good idea in order to not duplicate efforts. This can also act as a check that your implementation idea is sound. We are, for example, unlikely to accept contribs that require large modifications of the game directory structure.
- If your code is intended *primarily* as an example or shows a concept/principle rather than a working system, it is probably not suitable for `contrib/`. You are instead welcome to use it as part of a new tutorial!
- The code should ideally be contained within a single Python module. But if the contribution is large this may not be practical and it should instead be grouped in its own subdirectory (not as loose modules).
- The code must be well documented, including comments as well as a header in all modules. If a single file, the header should contain info about how to include the contrib in a game (installation instructions). If stored in a subdirectory, this info goes into a separate `README.md` file.
- The contribution should preferably be isolated (only make use of core Evennia) so it can easily be dropped into use. If it does depend on other contribs or third-party modules, these must be clearly documented and part of the installation instructions.
- The code itself should follow Evennia's Code style guidelines.
- Within reason, your contribution should be designed as genre-agnostic as possible. Limit the amount of game-style-specific code. Assume your code will be applied to a very different game than you had in mind when creating it.
- To make the licensing situation clear we assume all contributions are released with the same license as Evennia. If this is not possible for some reason, talk to us and we'll handle it on a case-by-case basis.

- While not *strictly* required at the current time, it's a big help if your contribution is covered by unit tests. Having unit tests will both help to make your code more stable and make sure small changes does not break it without it being noticed, it will also help us test its functionality and merge it quicker. If you contribution is a single module, you can add your unit tests to `evennia/contribs/tests.py`. If your contribution is bigger and in its own sub-directory you could just put the tests in your own `tests.py` file (Evennia will find it automatically).
- Merging of your code into Evennia is not guaranteed. Be ready to receive feedback and to be asked to make corrections or fix bugs. Furthermore, merging a contrib means the Evennia project takes on the responsibility of maintaining and supporting it. For various reasons this may be deemed to be beyond our manpower. However, if your code were to *not* be accepted for merger for some reason, we will instead add a link to your online repository so people can still find and use your work if they want.

CHAPTER 4

Soft Code

Softcode is a very simple programming language that was created for in-game development on TinyMUD derivatives such as MUX, PennMUSH, TinyMUSH, and RhostMUSH. The idea is that by providing a stripped down, minimalistic language for in-game use, you can allow quick and easy building and game development to happen without having to learn C/C++. There is an added benefit of not having to have to hand out shell access to all developers, and permissions can be used to alleviate many security problems.

Writing and installing softcode is done through a MUD client. Thus it is not a formatted language. Each softcode function is a single line of varying size. Some functions can be a half of a page long or more which is obviously not very readable nor (easily) maintainable over time.

Examples of Softcode

Here is a simple 'Hello World!' command:

```
@set me=HELLO_WORLD.C:$hello:@pemit %#=Hello World!
```

Pasting this into a MUX/MUSH and typing 'hello' will theoretically yield 'Hello World!', assuming certain flags are not set on your player object.

Setting attributes is done via `@set`. Softcode also allows the use of the ampersand (`&`) symbol. This shorter version looks like this:

```
&HELLO_WORLD.C me=$hello:@pemit %#=Hello World!
```

Perhaps I want to break the Hello World into an attribute which is retrieved when emitting:

```
&HELLO_VALUE.D me=Hello World
&HELLO_WORLD.C me=$hello:@pemit %#=[v(HELLO_VALUE.D)]
```

The `v()` function returns the `HELLO_VALUE.D` attribute on the object that the command resides (`me`, which is yourself in this case). This should yield the same output as the first example.

If you are still curious about how Softcode works, take a look at some external resources:

- http://www.tinymux.com/wiki/index.php/Softcode
- http://www.duh.com/discordia/mushman/man2x1

Problems with Softcode

Softcode is excellent at what it was intended for: *simple things*. It is a great tool for making an interactive object, a room with ambiance, simple global commands, simple economies and coded systems.
However, once you start to try to write something like a complex combat system or a higher end economy, you're likely to find yourself buried under a mountain of functions that span multiple objects across your entire code.

Not to mention, softcode is not an inherently fast language. It is not compiled, it is parsed with each calling of a function. While MUX and MUSH parsers have jumped light years ahead of where they once were they can still stutter under the weight of more complex systems if not designed properly.

To further illustrate the lack of readability for building larger systems in softcode, here is another example, PennMush softcode this time, for implementing a `+info` command. It allows you to store pages of extra character info that is later confirmed by admins and can be viewed by other players:

```
&INC`SET u(ifo)=@include u(ifo)/INC`TARGET;@include  \
u(ifo)/INC`FILENAME;@assert strlen(%q<filename>)=@nspemit \
%#=announce(INFO)%BERROR: Info file name empty.;@switch/inline \
gt(strlen(setr(attr,u(u(ifo)/FUN`FINDFILE,%q<target>,%q<filename>))),0)=1,{@assert \
or(isadmin(%#),strmatch(%q<target>,%#))=@nspemit \
%#=announce(INFO)%BERROR: You may not change another's Info \
files.;@switch/inline or(getstat(%q<target>/%q<attr>`FLAGS,Hidden),\
getstat(%q<target>/%q<attr> `FLAGS,Approved))=1,{@assert \
isadmin(%#)=@nspemit %#=announce(INFO)%BERROR: That Info File may not \
be changed by you.}},0,{@break gt(strlen(%q<filename>),18)=@nspemit \
%#=ERROR: Info names are limited to 18 characters or less.;@break \
regmatchi(%q<filename>,\\|)=@nspemit %#=ERROR: Pipe symbols are not \
allowed in info names.;@break regmatchi(%q<filename>,\/)=@nspemit \
%#=ERROR: Slashes symbols are not allowed in info names.;@assert \
strlen(%1)=ERROR: Text field empty. To delete an +info file, use \
+info/delete.};&[strfirstof(%q<attr>,setr(attr,D`INFOFILE`[nextslot(%q<target>,\
D`INFOFILE)]))] %q<target>=%q<filename>;&%q<attr>`CONTENTS %q<target>=%1;th \
setstat(%q<target>/%q<attr>`FLAGS,SetBy,%#);th \
setstat(%q<target>/%q<attr>`FLAGS,SetOn,secs());@switch/inline \
strmatch(%#,%q<target>)=1,{@nspemit %#=announce(INFO)%BYou set your \
%q<filename> Info File},{@nspemit %#=announce(INFO)%BYou set \
[name(%q<target>)]'s %q<filename> Info File!;@nspemit \
%q<target>=announce(INFO)%B%n set your %q<filename> Info File!}
```

(Note that the softcode is actually all one line, it was split to be viewable on this wiki). Below is the rough Evennia equivalent functionality as an Evennia command method, originally written by the same softcode author after a week of learning Evennia:

```
def switch_set(self, target, files, rhs, isadmin):
    caller = self.caller
    if caller is not target and not isadmin:
        caller.msg("ERROR: You may not set that person's files.")
        return
    if not self.rhs:
        caller.msg("ERROR: No info file contents entered to set.")
        return
```

```
    for info in files:
        inf = info.lower().strip()
        if not re.match('^[\w-]+$', inf):
            caller.msg("ERROR: File '%s' could not be set: "
                       "may only use alphanumeric characters, -,"
                       "and spaces in info names." % info)
        elif self.files.get(inf, {}).get("approved", None):
            caller.msg("ERROR: File '%s' could not be set: "
                       "file is approved." % info)
        else:
            self.files[inf] = {"contents":rhs,
                               "setby":caller,
                               "seton":"timestamp",
                               "displayname":info}
            if target is caller:
                caller.msg("Info File '%s' set!" % info)
            else:
                caller.msg("Info File '%s' set!" % info)
                target.msg("%s set your %s info file!" % \
                                        (caller.key, info)
            target.db.infofiles = dict(self.files)
```

The details of the implementation are unimportant, the main point is the difference in readability (and, by extension, maintainability).

Changing Times

Now that starting text-based games is easy and an option for even the most technically inarticulate, new projects are a dime a dozen. People are starting new MUDs every day with varying levels of commitment and ability. Because of this shift from fewer, larger, well-staffed games to a bunch of small, one or two developer games, some of the benefit of softcode fades.

Softcode is great in that it allows a mid to large sized staff all work on the same game without stepping on one another's toes. As mentioned before, shell access is not necessary to develop a MUX or a MUSH. However, now that we are seeing a lot more small, one or two-man shops, the issue of shell access and stepping on each other's toes is a lot less.

Our Solution

Evennia shuns in-game softcode for on-disk Python modules. Python is a popular, mature and professional programming language. You code it using the conveniences of modern text editors. Evennia developers have access to the entire library of Python modules out there in the wild - not to mention the vast online help resources available. Python code is not bound to one-line functions on objects but complex systems may be organized neatly into real source code modules, sub-modules, or even broken out into entire Python packages as desired.

So what is *not* included in Evennia is a MUX/MOO-like online player coding system. Advanced coding in Evennia is primarily intended to be done outside the game, in full-fledged Python modules. Advanced building is best handled by extending Evennia's command system with your own sophisticated building commands. We feel that with a small development team you are better off using a professional source-control system (svn, git, bazaar, mercurial etc) anyway.

Your Solution

Adding advanced and flexible building commands to your game is easy and will probably be enough to satisfy most creative builders. However, if you really, *really* want to offer online coding, there is of course nothing stopping you from adding that to Evennia, no matter our recommendations. You could even re-implement MUX' softcode in Python should you be very ambitious.

CHAPTER 5

Using MUX as a Standard

Evennia allows for any command syntax. If you like the way DikuMUDs, LPMuds or MOOs handle things, you could emulate that with Evennia. If you are ambitious you could even design a whole new style, perfectly fitting your own dreams of the ideal game.

We do offer a default however. The default Evennia setup tends to *resemble* MUX2, and its cousins PennMUSH, TinyMUSH, and RhostMUSH. While the reason for this similarity is partly historical, these codebases offer very mature feature sets for administration and building.

Evennia is *not* a MUX system though. It works very differently in many ways. For example, Evennia deliberately lacks an online softcode language (a policy explained on our softcode policy page). Evennia also does not shy from using its own syntax when deemed appropriate: the MUX syntax has grown organically over a long time and is, frankly, rather arcane in places. All in all the default command syntax should at most be referred to as "MUX-like" or "MUX-inspired".

Documentation policy

All the commands in the default command sets should have their doc-strings formatted on a similar form:

```
"""
Short header

Usage:
  key[/switches, if any] <mandatory args> [<optional args or types>]

Switches:
  switch1    - description
  switch2    - description

Examples:
  usage example and output

Longer documentation detailing the command.
```

```
"""
```

Two spaces are used for indentation in all default commands. As per standard computer convention, square brackets `[ ]` surround *optional arguments* whereas angled brackets `< >` surround a *description* of what to write rather than the exact syntax. The `Switches` and `Examples` blocks are added as required by the Command.

Here is the `nick` command as an example:

```
"""
Define a personal alias/nick

Usage:
  nick[/switches] <nickname> = [<string>]
  alias              ''

Switches:
  object   - alias an object
  player   - alias a player
  clearall - clear all your aliases
  list     - show all defined aliases (also "nicks" works)

Examples:
  nick hi = say Hello, I'm Sarah!
  nick/object tom = the tall man

A 'nick' is a personal shortcut you create for your own use [...]

  """
```

For commands that *require arguments*, the policy is for it to return a `Usage:` string if the command is entered without any arguments. So for such commands, the Command body should contain something to the effect of

```
if not self.args:
    self.caller.msg("Usage: nick[/switches] <nickname> = [<string>]")
    return
```

CHAPTER 6

Game Planning

So you have Evennia up and running. You have a great game idea in mind. Now it's time to start cracking! But where to start? Here are some ideas for a workflow. Note that the suggestions on this page are just that - suggestions. Also, they are primarily aimed at a lone hobby designer or a small team developing a game in their free time. There is an article in the Imaginary Realities e-zine which was written by the Evennia lead dev. It focuses more on you finding out your motivations for making a game - you can read the article here.

Below are some minimal steps for getting the first version of a new game world going with players. It's worth to at least make the attempt to do these steps in order even if you are itching to jump ahead in the development cycle. On the other hand, you should also make sure to keep your work fun for you, or motivation will falter. Making a full game is a lot of work as it is, you'll need all your motivation to make it a reality.

Remember that *99.99999% of all great game ideas never lead to a game*. Especially not to an online game that people can actually play and enjoy. So our first all overshadowing goal is to beat those odds and get *something* out the door! Even if it's a scaled-down version of your dream game, lacking many "must-have" features! It's better to get it out there and expand on it later than to code in isolation forever until you burn out, lose interest or your hard drive crashes.

Like is common with online games, getting a game out the door does not mean you are going to be "finished" with the game - most MUDs add features gradually over the course of years - it's often part of the fun!

Planning (step 1)

This is what you do before having coded a single line or built a single room. Many prospective game developers are very good at *parts* of this process, namely in defining what their world is "about": The theme, the world concept, cool monsters and so on. It is by all means very important to define what is the unique appeal of your game. But it's unfortunately not enough to make your game a reality. To do that you must also have an idea of how to actually map those great ideas onto Evennia.

A good start is to begin by planning out the basic primitives of the game and what they need to be able to do. Below are a far-from-complete list of examples (and for your first version you should definitely try for a much shorter list):

Systems

These are the behind-the-scenes features that exist in your game, often without being represented by a specific in-game object.

- Should your game rules be enforced by coded systems or are you planning for human game masters to run and arbitrate rules?
- What are the actual mechanical game rules? How do you decide if an action succeeds or fails? What "rolls" does the game need to be able to do? Do you base your game off an existing system or make up your own?
- Does the flow of time matter in your game - does night and day change? What about seasons? Maybe your magic system is affected by the phase of the moon?
- Do you want changing, global weather? This might need to operate in tandem over a large number of rooms.
- Do you want a game-wide economy or just a simple barter system? Or no formal economy at all?
- Should characters be able to send mail to each other in-game?
- Should players be able to post on Bulletin boards?
- What is the staff hierarchy in your game? What powers do you want your staff to have?
- What should a Builder be able to build and what commands do they need in order to do that?
- etc.

Rooms

Consider the most basic room in your game.

- Is a simple description enough or should the description be able to change (such as with time, by light conditions, weather or season)?
- Should the room have different statuses? Can it have smells, sounds? Can it be affected by dramatic weather, fire or magical effects? If so, how would this affect things in the room? Or are these things something admins/game masters should handle manually?
- Can objects be hidden in the room? Can a person hide in the room? How does the room display this?
- etc.

Objects

Consider the most basic (non-player-controlled) object in your game.

- How numerous are your objects? Do you want large loot-lists or are objects just role playing props created on demand?
- Does the game use money? If so, is each coin a separate object or do you just store a bank account value?
- What about multiple identical objects? Do they form stacks and how are those stacks handled in that case?
- Does an object have weight or volume (so you cannot carry an infinite amount of them)?
- Can objects be broken? If so, does it have a health value? Is burning it causing the same damage as smashing it? Can it be repaired?
- Is a weapon a specific type of object or are you supposed to be able to fight with a chair too? Can you fight with a flower or piece of paper as well?
- NPCs/mobs are also objects. Should they just stand around or should they have some sort of AI?

- Are NPCs/mobs differet entities? How is an Orc different from a Kobold, in code - are they the same object with different names or completely different types of objects, with custom code?
- Should there be NPCs giving quests? If so, how would you track quest status and what happens when multiple players try to do the same quest? Do you use instances or some other mechanism?
- etc.

Characters

These are the objects controlled directly by Players.

- Can players have more than one Character active at a time or are they allowed to multi-play?
- How does a Player create their Character? A Character-creation screen? Answering questions? Filling in a form?
- Do you want to use classes (like "Thief", "Warrior" etc) or some other system, like Skill-based?
- How do you implement different "classes" or "races"? Are they separate types of objects or do you simply load different stats on a basic object depending on what the Player wants?
- If a Character can hide in a room, what skill will decide if they are detected?
- What skill allows a Character to wield a weapon and hit? Do they need a special skill to wield a chair rather than a sword?
- Does a Character need a Strength attribute to tell how much they can carry or which objects they can smash?
- What does the skill tree look like? Can a Character gain experience to improve? By killing enemies? Solving quests? By roleplaying?
- etc.

A MUD's a lot more involved than you would think and these things hang together in a complex web. It can easily become overwhelming and it's tempting to want *all* functionality right out of the door. Try to identify the basic things that "make" your game and focus *only* on them for your first release. Make a list. Keep future expansions in mind but limit yourself.

Coding (step 2)

This is the actual work of creating the "game" part of your game. Many "game-designer" types tend to gloss over this bit and jump directly to **World Building**. Vice versa, many "game-coder" types tend to jump directly to this part without doing the **Planning** first. Neither way is good and *will* lead to you having to redo all your hard work at least once, probably more.

Evennia's Developer Central tries to help you with this bit of development. We also have a slew of Tutorials with worked examples. Evennia tries hard to make this part easier for you, but there is no way around the fact that if you want anything but a very basic Talker-type game you *will* have to bite the bullet and code your game (or find a coder willing to do it for you).

Even if you won't code anything yourself, as a designer you need to at least understand the basic paradigms of Evennia, such as Objects, Commands and Scripts and how they hang together. We recommend you go through the Tutorial World in detail (as well as glancing at its code) to get at least a feel for what is involved behind the scenes. You could also look through the tutorial for building a game from scratch.

During Coding you look back at the things you wanted during the **Planning** phase and try to implement them. Don't be shy to update your plans if you find things easier/harder than you thought. The earlier you revise problems, the easier they will be to fix.

A good idea is to host your code online (publicly or privately) using version control. Not only will this make it easy for multiple coders to collaborate (and have a bug-tracker etc), it also means your work is backed up at all times. Here are instructions for setting up a sane developer environment with proper version control.

"Tech Demo" Building

This is an integral part of your Coding. It might seem obvious to experienced coders, but it cannot be emphasized enough that you should *test things on a small scale* before putting your untested code into a large game-world. The earlier you test, the easier and cheaper it will be to fix bugs and even rework things that didn't work out the way you thought they would. You might even have to go back to the **Planning** phase if your ideas can't handle their meet with reality.

This means building singular in-game examples. Make one room and one object of each important type and test so they work correctly in isolation. Then add more if they are supposed to interact with each other in some way. Build a small series of rooms to test how mobs move around … and so on. In short, a test-bed for your growing code. It should be done gradually until you have a fully functioning (if not guaranteed bug-free) miniature tech demo that shows *all* the features you want in the first release of your game. There does not need to be any game play or even a theme to your tests, this is only for you and your co-coders to see. The more testing you do on this small scale, the less headaches you will have in the next phase.

World Building (step 3)

Up until this point we've only had a few tech-demo objects in the database. This step is the act of populating the database with a larger, thematic world. Too many would-be developers jump to this stage too soon (skipping the **Coding** or even **Planning** stages). What if the rooms you build now doesn't include all the nice weather messages the code grows to support? Or the way you store data changes under the hood? Your building work would at best require some rework and at worst you would have to redo the whole thing. And whereas Evennia's typeclass system does allow you to edit the properties of existing objects, some hooks are only called at object creation … Suffice to say you are in for a *lot* of unnecessary work if you build stuff en masse without having the underlying code systems in some reasonable shape first.

So before starting to build, the "game" bit (**Coding + Testing**) should be more or less **complete**, *at least to the level of your initial release*.

Before starting to build, you should also plan ahead again. Make sure it is clear to yourself and your eventual builders just which parts of the world you want for your initial release. Establish for everyone which style, quality and level of detail you expect. Your goal should *not* be to complete your entire world in one go. You want just enough to make the game's "feel" come across. You want a minimal but functioning world where the intended game play can be tested and roughly balanced. You can always add new areas later.

During building you get free and extensive testing of whatever custom build commands and systems you have made at this point. Since Building often involves different people than those Coding, you also get a chance to hear if some things are hard to understand or non-intuitive. Make sure to respond to this feedback.

Alpha Release

As mentioned, don't hold onto your world more than necessary. *Get it out there* with a huge *Alpha* flag and let people try it! Call upon your alpha-players to try everything - they *will* find ways to break your game in ways that you never could have imagined. In Alpha you might be best off to focus on inviting friends and maybe other MUD developers, people who you can pester to give proper feedback and bug reports (there *will* be bugs, there is no way around it). Follow the quick instructions for Online Setup to make your game visible online. If you hadn't already, make sure to

put up your game on the Evennia game index so people know it's in the works (actually, even pre-alpha games are allowed in the index so don't be shy)!

Beta Release/Perpetual Beta

Once things stabilize in Alpha you can move to *Beta* and let more people in. Many MUDs are in perpetual beta, meaning they are never considered "finished", but just repeat the cycle of Planning, Coding, Testing and Building over and over as new features get implemented or Players come with suggestions. As the game designer it is now up to you to gradually perfect your vision.

Congratulate yourself!

You are worthy of a celebration since at this point you have joined the small, exclusive crowd who have made their dream game a reality!

CHAPTER 7

Installation and setup

This chapter helps with installing and managing the server itself as well as figuring out where things go.

Getting Started

This will help you download, install and start Evennia for the first time.

> Note: You don't need to make anything visible to the 'net in order to run and test out Evennia. Apart from downloading and updating you don't even need an internet connection until you feel ready to share your game with the world.

- *Quick Start*
- *Requirements*
- *Linux Install*
- *Mac Install*
- *Windows Install*
- Running in Docker
- *Where to go next*
- *Troubleshooting*

Quick start

For the impatient. If you have trouble with a step, you should jump on to the more detailed instructions for your platform.

1. Install Python, GIT and python-virtualenv. Start a Console/Terminal.
2. `cd` to some place you want to do your development (like a folder `/home/anna/muddev/` on Linux or a folder in your personal user directory on Windows).

3. `virtualenv pyenv`
4. `source pyenv/bin/activate` (Linux, Mac), `pyenv\Scripts\activate` (Windows)
5. `git clone https://github.com/evennia/evennia.git`
6. `pip install -e evennia`
7. `evennia --init mygame`
8. `cd mygame`
9. `evennia migrate`
10. `evennia start` (make sure to make a superuser when asked) Evennia should now be running and you can connect to it by pointing a web browser to `http://localhost:8000` or a MUD telnet client to `localhost:4000` (use `127.0.0.1` if your OS does not recognize `localhost`).

Requirements

Any system with Python support should work.

- Linux/Unix
- Windows (2000, XP, Vista, Win7, Win8, Win10)
- Mac OSX (>=10.5 recommended)
- Python (v2.7+, not supporting v3.x).
- Pip. Python installer, included with Python 2.7.9+ but can also be installed separately.
- virtualenv for making isolated Python environments. Installed with `pip install virtualenv`.
- Linux/Mac users will need the `gcc` and `python-dev` packages or equivalent.
- Windows users need MS Visual C++ and pypiwin32.
- GIT - version control software for getting and updating Evennia itself
- Mac users can use the git-osx-installer or the MacPorts version.
- Twisted (v16.0+)
- ZopeInterface (v3.0+) - usually included in Twisted packages
- Windows users need pypiwin32.
- Windows users need MS Visual C++.
- Django (v1.9+, be warned that latest dev version is usually untested with Evennia)
- Pillow (Python Image Library). This is often distributed with Django. As a backup you can also try the older PIL, on which Pillow is based.

Linux install

If you run into any issues during the installation and first start, please check out **'Linux troubleshooting'_**. Also, one of our devs made a Linux install video, check it out!

For Debian-derived systems (like Ubuntu, Mint etc), start a terminal and install the *dependencies*:

```
sudo apt-get install python python-dev git python-pip python-virtualenv gcc
```

You should make sure to *not* be `root` after this step, running as `root` is a security risk. If your Linux distro defaults to Python3 you need to install Python2.7+ explicitly (Evennia does not support Python3 at this time). Next create a folder where you want to do all your Evennia development and start a virtualenv inside:

```
mkdir muddev
cd muddev
# if your linux defaults to python2:
virtualenv pyenv
# if your linux defaults to python3:
virtualenv -p /usr/bin/python2.7 pyenv
```

Using a virtualenv is good Python practice. A new folder `pyenv` will appear. This folder will hold a self-contained setup of Python packages without interfering with default Python packages on your system or the Linux distro lagging behind on Python package versions. Activate the virtual environment:

```
source pyenv/bin/activate
```

The text `(pyenv)` should appear next to your prompt to show the virtual environment is active. You need to activate the virtualenv like this *every time* you start a new terminal to get access to the Evennia install. Next we fetch Evennia itself:

```
git clone https://github.com/evennia/evennia.git
pip install -e evennia
```

A new folder `evennia` will appear containing the Evennia library. If install failed with any issues, see **'Linux Troubleshooting'_**. Next we'll start our new game, here called "mygame", which will create yet another new folder where you will be creating your new game:

```
evennia --init mygame
```

You can configure Evennia extensively, for example for using a different database. For now we'll just stick to the defaults though.

```
cd mygame
evennia migrate   # (make sure to create a superuser when asked. Email is optional.)
evennia start
```

Your game should now be running! Open a web browser at `http://localhost:8000` or point a telnet client to `localhost:4000` and log in with the user you created. Check out *where to go next*.

Mac install

The Evennia server is a terminal program. Open the terminal e.g. from *Applications->Utilities->Terminal*. Here is an introduction to the Mac terminal if you are unsure how it works. If you run into any issues during the installation, please check out **'Mac troubleshooting'_**.

- Python should already be installed. (This discusses how you may upgrade it). Remember that uou need Python2.7.x, not Python3+!
- GIT can be obtained with git-osx-installer or via MacPorts as described here.
- If your Python version is lower than 2.7.9 you will need to install `pip` manually with `sudo easy_install pip`.
- Fetch virtualenv with `pip install virtualenv`.
- If you run into issues with installing `Twisted` later you may need to install gcc and the python headers.

Next create a folder where you want to do all your Evennia development and start a virtualenv inside:

```
mkdir muddev
cd muddev
# if your mac defaults to python2:
virtualenv pyenv
# if your mac defaults to python3:
virtualenv -p /path/to/your/python2.7 pyenv
```

Using a virtualenv is a good Python practice. A new folder `pyenv` will appear. This folder will hold a self-contained setup of Python packages without interfering with default Python packages on your system. Activate the virtual environment:

```
source pyenv/bin/activate
```

The text `(pyenv)` should appear next to your prompt to show the virtual environment is active. You need to activate the virtualenv like this *every time* you start a new terminal to get access to the Evennia install. Next we fetch Evennia itself:

```
git clone https://github.com/evennia/evennia.git
pip install -e evennia
```

If install failed with any issues, see **'Mac Troubleshooting'_**. Next we'll start our new game. We'll call it "mygame" here. This creates a new folder where you will be creating your new game:

```
evennia --init mygame
```

You can configure Evennia extensively, for example to use a different database. We'll use the defaults here.

```
cd mygame
evennia migrate   (make sure to create a superuser when asked. Email is optional.)
evennia start
```

Your game should now be running! Open a web browser at `http://localhost:8000` or point a telnet client to `localhost:4000` and log in with the user you created. Check out *where to go next*.

Windows install

If you run into any issues during the installation, please check out **'Windows troubleshooting'_**.

The Evennia server itself is a command line program. In the Windows launch menu, start *All Programs -> Accessories -> command prompt* and you will get the Windows command line interface. Here is one of many tutorials on using the Windows command line if you are unfamiliar with it.

- Install Python from the Python homepage. You will need to be a Windows Administrator to install packages. You want Python version 2.7+ (Evennia does not support Python3), ideally 2.7.9 or later. You should usually get the 64bit-version if your system supports it. When installing, make sure to check-mark *all* install options, especially the one about making Python available on the path (you may have to scroll to see it). This allows you to just write `python` in any console without first finding where the `python` program actually sits on your hard drive.
- You need to also get GIT and install it. You can use the default install options but when you get asked to "Adjust your PATH environment", you should select the second option "Use Git from the Windows Command Prompt", which gives you more freedom as to where you can use the program.
- Finally you should install the Microsoft Visual C++ compiler for Python as well as the pypiwin32 python headers.

First create a new folder for all your Evennia development (let's call it `muddev`), then go to it in the console and start a virtualenv inside:

```
cd path\to\muddev
pip install virtualenv
# if your setup defaults to Python2.x (most likely)
virtualenv pyenv
# if your setup defaults to Python3+
virtualenv -p C:\Python27\python.exe pyenv
```

Using a virtualenv is a standard Python practice. A new folder `pyenv` will appear. This folder will hold a self-contained setup of Python packages without interfering with default Python packages on your system. Activate the virtual environment by running the script:

```
# If you are using a standard command prompt, you can use the following
.\pyenv\scripts\activate
# If you are using a PS Shell, Git Bash, or other, you can use the following
pyenv/scripts/activate.bat
```

The text `(pyenv)` should appear next to your prompt to show the virtual environment is active. You need to activate the virtualenv like this *every time* you start a new terminal to get access to the Evennia install. Next we fetch Evennia itself:

```
git clone https://github.com/evennia/evennia.git
pip install -e evennia
```

If everything went well, Evennia is installed. If the install failed with any issues, see **'Windows Troubleshooting'_**. Next we'll start our new game, we'll call it "mygame" here. This creates a new folder where you will be creating your new game:

```
evennia --init mygame
```

You can configure Evennia extensively, for example for using a different database. We'll use the defaults here.

```
cd mygame
evennia migrate   (make sure to create a superuser when asked. Email is optional.)
evennia start
```

Your game should now be running! Open a web browser at `http://localhost:8000` or point a telnet client to `localhost:4000` and log in with the user you created. Check out *where to go next*.

Where to next

Welcome to Evennia! Your new game is, once you logged in, fully functioning but empty. To get started, follow the instructions in the `Limbo` room to create Evennia's tutorial world - it's a small solo quest to explore.

Once you get back to `Limbo` (if you get stuck in the tutorial quest you can do `@tel #2` to jump to Limbo), a good idea is to learn how to start, stop and reload the Evennia server. After that, why not experiment with creating some new items and build some new rooms out from Limbo.

From here on, you could move on to do one of our introductory tutorials or simply dive headlong into Evennia's comprehensive manual. If you have any questions, you can always ask in the developer chat `#evennia` on `irc.freenode.net` or by posting to the Evennia forums. Finally, if you are itching to help out or support Evennia (awesome!) have an issue to report or a feature to request, see here.

Enjoy your stay!

Troubleshooting

If you have issues with installing or starting Evennia for the first time, check the section for your operating system below. If you have an issue not covered here, please report it so it can be fixed or a workaround found!

Linux troubleshooting

- If you get an error when installing Evennia (especially with lines mentioning failing to include `Python.h`) then try `sudo apt-get install python-setuptools`. Once installed, run `pip install -e evennia` again.
- Under some not-updated Linux distributions you may run into errors with a too-old `setuptools` or missing `functools`. If so, update your environment with `pip install --upgrade pip wheel setuptools`. Then try `pip install evennia` again.
- A common error on Ubuntu 16 and others is that you can't start the server but get an error saying "*Twisted requires zope.interface 3.6.0 or later: no module named zope.interface.*". This happens even though a much later version of `zope.interface` is clearly installed in the virtualenv (as verified with `pip list`). This appears to be due to a bug in the zope installer; it fails to install an empty `__init__.py` file. To fix this, issue the following command: `touch pyenv/local/lib/python2.7/site-packages/zope/__init__.py`. That creates the file and things should then work correctly henceforth.
- One user reported a rare issue on Ubuntu 16 is an install error on installing Twisted; `Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build-vnIFTg/twisted/` with errors like `distutils.errors.DistutilsError: Could not find suitable distribution for Requirement.parse('incremental>=16.10.1')`. This appears possible to solve by simply updating Ubuntu with `sudo apt-get update && sudo apt-get dist-upgrade`.
- Users of Fedora (notably Fedora 24) has reported a `gcc` error saying the directory `/usr/lib/rpm/redhat/redhat-hardened-cc1` is missing, despite `gcc` itself being installed. The confirmed work-around seems to be to install the `redhat-rpm-config` package with e.g. `sudo dnf install redhat-rpm-config`.

Mac troubleshooting

- Some mac users have reported not being able to connect to `localhost` (i.e. your own computer). If so, try to connect to `127.0.0.1` instead, which is the same thing. Use port 4000 from mud clients and port 8000 from the web browser as usual.

Windows troubleshooting

- If your MUD client cannot connect to `localhost:4000`, try the equivalent `127.0.0.1:4000` instead. Some MUD clients on Windows does not appear to understand the alias `localhost`.
- If you run `virtualenv pyenv` and get a `'virtualenv' is not recognized as an internal or external command, operable program or batch file.` error, you can `mkdir pyenv` then `python -m virtualenv .` as a workaround.
- If you are using an older version of Twisted than 16.x or if you are using an older version of Evennia with the latest version of Twisted you may get an error at startup. The error is due to the Twisted executable changing names on Windows from `twistd.py` to `twistd.exe`. If you don't want to reinstall evennia anew you can instead edit `evennia\evennia\server\twistd.bat`. This is a file containing a single line of text. Change the single occurrence of `twistd.py` to instead read `twistd.exe`. Save. Evennia should now start correctly.

- Some Windows users get an error installing the Twisted 'wheel'. A wheel is a pre-compiled binary package for Python. A common reason for this error is that you are using a 32-bit version of Python, but Twisted has not yet uploaded the latest 32-bit wheel. Easiest way to fix this is to install a slightly older Twisted version. So if, say, version `16.1` failed, install `16.0` manually with `pip install twisted==16.0`. If that doesn't work either, you could also try to get a 64-bit version of Python. If so, you should `deactivate` the virtualenv, delete the `pyenv` folder and recreate it anew.

Start Stop Reload

You control Evennia from your game folder (we refer to it as `mygame/` here), using the `evennia` program. If the `evennia` program is not available on the command line you must first install Evennia as described in the Getting Started page.

A common reason for not seeing the `evennia` command is to forget to (re)start the virtual environment (in a folder called `pyenv` if you followed the Getting Started page). You need to do this every time you start a new terminal/console and should see `(pyenv)` at the beginning of the line. The virtualenv allows to install all Python dependencies without needing root or disturbing the global packages in the repo (which are often older).

```
source pyenv/bin/activate (Linux/Unix)
pyenv/Scripts/activate` (Windows)
```

Below are described the various management options. Run

```
evennia -h
```

to give you a brief help and

```
evennia menu
```

to give you a menu with options.

Starting Evennia

Evennia consists of two components, the Evennia Server and Portal. Briefly, the *Server* is what is running the mud. It handles all game-specific things but doesn't care exactly how players connect, only that they have. The *Portal* is a gateway to which players connect. It knows everything about telnet, ssh, webclient protocols etc but very little about the game. Both are required for a functioning mud.

```
evennia start
```

The above command automatically starts both Portal and Server at the same time. The Server will log to the terminal (`stdout`), while the Portal will log to its log file in `mygame/server/log`. This is the most useful mode for development since you see logged errors directly in the terminal.

You can also start the two components one at a time:

```
evennia start server
evennia start portal
```

You can also start the server in *interactive mode* with the `-i` flag:

```
evennia -i start
```

This will start Evennia as a foreground process. You can then stop it completely with `Ctrl-C`.

Reloading

The act of *reloading* means the *Server* program is shut down and then restarted again. With the default commands you initiate a reload with the `@reload` command from inside the game. Everyone will get a message and the game will be briefly paused for all players as the server reboots. Since they are connected to the *Portal*, their connections are not lost.

Reloading is as close to a "warm reboot" you can get. It reinitializes all code of Evennia, but doesn't kill "persistent" scripts. It also calls `at_server_reload()` hooks on all objects so you can save eventual temporary properties you want.

You can also reload the server from outside the game:

```
evennia reload
```

This is very useful if you are testing new functionality and introduce some critical error that makes it impossible for Evennia to load your module. If the module in question is, for example, describing your in-game Character, this may mean that you can't enter commands in-game anymore. A terminal-line `evennia reload` (or `evennia start` if the server itself couldn't recover from the error) will get everything going again once the bug in your code is fixed.

> External reloading from the command line is not supported on the Windows platform. This is due to limitations in Windows signal handling. To reload the server on Windows, use `@reload` from in-game. You can also use `evennia stop && evennia start` for a cold restart. A cold restart will kick everyone but may be necessary if you introduced a syntax error causing the in-game `@reload` command to not be possible to load.

Resetting

Resetting is the equivalent of a "cold reboot" of the Server component - it will restart but will behave as if it was fully shut down. You initiate a reset using the `@reset` command from inside the game. As opposed to a "real" shutdown, no players will be disconnected during a reset. A reset will however purge all non-persistent scripts and will call `at_server_shutdown()` hooks. It can be a good way to clean unsafe scripts during development, for example.

A reset is equivalent to

```
evennia stop server
evennia start server
```

Shutting down

A full shutdown closes Evennia completely, both Server and Portal. All players will be booted and systems saved and turned off cleanly. From inside the game you initiate a shutdown with the `@shutdown` command.

From command line you do

```
evennia stop
```

You will see messages of both Server and Portal closing down. All players will see the shutdown message and then be disconnected. The same effect happens if you press `Ctrl+C` while the server runs in interactive mode.

> If you run Windows you will be asked "Terminate batch job (Y/N)?" multiple times in a row. This is an annoying practice of Windows console that can't be turned off. Instead of entering `Y` over and over you can press `Ctrl+C` once to bypass the questions. There are supposedly third-party replacements for the Windows Console that allows to turn this off, report your findings!

Django options

The `evennia` program will also pass-through options used by the `django-admin`. These operate on the database in various ways.

```
evennia migrate # migrate the database
evennia shell   # launch an interactive, django-aware python shell
evennia dbshell # launch database shell
```

For (many) more options, see the django-admin docs.

Advanced handling of Evennia processes

If you should need to manually manage Evennia's processors (or view them in a task manager program such as Linux' `top` or the more advanced `htop`), you will find the following processes to be related to Evennia:

- 1 x `evennia_runner.py` - This is a process started by the `evennia` launcher. The `evennia_runner.py` instance watches the Server and Portal processes respectively (but commonly only the Server). When the other process shuts down the runner will check if a "restart" is desired and if so restart it again. This is how reloading the server works.
- 2 x `twistd ... server.py` - One of these processes manages Evennia's Server component, the main game. The second process (with the same name but different process id) handle's Evennia's internal web server. You can look at `mygame/server/server.pid` to determine which is the main process of the two.
- 2 x `twistd ... portal.py` - One of these processes maintains the Portal component and all external-facing protocols. The second process (with the same name but a different process id) handles the AMP client that communicates data between the Portal and the Server. You can look at `mygame/server/portal.pid` to determine which is the main process of the two.

Syntax errors during live development

During development, you will usually modify code and then reload the server to see your changes. This is done by Evennia re-importing your custom modules from disk. Usually bugs in a module will just have you see a traceback in the game, in the log or on the command line. For some really serious syntax errors though, your module might not even be recognized as valid Python. Evennia may then fail to restart correctly.

From inside the game you see a text about the Server restarting followed by an ever growing list of "...". Usually this only lasts a very short time (up to a few seconds). If it seems to go on, it means the Portal is still running (you are still connected to the game) but the Server-component of Evennia failed to restart (that is, it remains in a shut-down state). Look at your log files or terminal to see what the problem is - you will usually see a clear traceback showing what went wrong.

Fix your bug then run

```
evennia start
```

Assuming the bug was fixed, this will start the Server manually (while not restarting the Portal). In-game you should now get the message that the Server has successfully restarted.

Recovering from critical shutdowns

If your server died unexpectedly it may not have had time to clean up after itself correctly. This can happen if you shut down your computer with Evennia running or force-killed the process (such as with `killall -9 twistd`

under Linux). What then happens is that two small files that should have been deleted remain behind. These are `mygame/server/server.pid` and `mygame/server/portal.pid`.

You notice this by Evennia telling you that the Server/Portal is "already running" despite it clearly does not. First try to stop it manually:

```
evennia stop
```

If there is a problem Evennia will tell you it cannot signal the system to close and that the pid files are "stale". Fixing this is simple - just delete `portal.pid` and/or `server.pid` and start anew.

Optional: Server startup script (Linux only)

This is considered an advanced section. First make sure Evennia starts normally.

If you start Evennia on a remote server and then disconnect you will likely find that `@reload` (or `evennia reload`) no longer works. The server will shut down and just not come back up again. The reason for this is that when you started Evennia, a silent little program called the *runner* also started in the background. The runner's job is to catch the Server shutdown and kick it back into gear if you wanted a reload. If you kill the terminal you will also kill this little runner program and thus the Server can't restart.

Enter Gnu Screen. Screen is a standard Linux/Unix program for managing terminal sessions. If you control the server you can install it (on Debian derivatives) with `apt-get install screen`. Most remote server solutions will offer screen by default or you should easily be able to request it to be installed. When you are running under Screen, logging off will not kill the terminal but just *disconnect* from it - it (and the runner) will keep running happily without you. You can later "attach" to it again and continue where you were.

Evennia supplies a bash script for managing the server under Screen. It assumes you installed Evennia according to the Getting Started instructions and have it running already. The script is found in the evennia repository's top level, as `bin/unix/evennia-screen.sh`. This is how you use it:

1. Copy `evennia-screen.sh` to some place in `mygame`, for example `mygame/server` (but it could really go anywhere).
2. Open the copied file in a text editor and change the variables `GAMENAME`, `VIRTUALENV` and `GAMEDIR` to fit your game setup. You should use absolute paths.
3. Make it executable by you with `chmod u+x evennia-screen.sh`

Make sure your game is not running and then start Evennia using `./evennia-screen.sh start`. Apart from a short message nothing will appear to happen. But behind the scenes a new Screen session was started and Evennia launched inside it. To see the availabe Screen sessions, enter this in the terminal:

```
screen -ls
```

You should get back some lines including this one:

```
19989.mygame    (12/08/2016 10:31:31 PM)        (Detached)
```

The exact numbers will vary, but the `GAMENAME` you entered in the script should be seen as the name of the session ("mygame" here). Now we connect to the session:

```
screen -r mygame
```

Boom - you are now inside the Screen session and should see the Evennia startup messages. You are already inside the virtualenv and can operate the server normally. If you close the window, the Screen session will just "detach". To detach without logging out, press `Ctrl-a d` (that is, hold `Ctrl-a` and then press `d`). Actually killing Screen is done by `Ctrl-d` (the normal Unix disrupt signal).

Screen is a powerful program with a lot of very useful features. you are wise to read up more on it if you work remotely on any server.

Optional: init.d startup script (Linux only)

This is considered an advanced section.

> For the final steps of this section you will need root access to your server, or be able to ask someone with root access to help you. You will also need a working `evennia-screen.sh` script from the previous section, so make sure to do that first.

When running on a remote server you may want Evennia to start automatically when the server starts. This could be important for unexpected power outages for example. You may also want to run the server as an isolated user for security reasons.

Evennia supplies a bash script for use with the common System V init system (if you have `/etc/init.d` this is what you use. This was tested with Debian-derived distros). The script is found at the root of the evennia repository, as `bin/unix/evennia-screen-initd.sh`.

To use the script, do the following:

1. Copy `bin/unix/evennia-screen.sh` to a temporary place so you can edit it without modifying the original.
2. Edit the script and change the lines for `SCRIPTPATH` and `USER` to match your setup. `SCRIPTPATH` is the absolute path to your working `evennia-screen.sh` script from the previous section. The `USER` must have access to launching Evennia. It's commonly your own user name but you might also consider creating a separate low-privilege account only for Evennia. *Obs: Never put ''USER'' to "root", Evennia should never be run as root!*.

If you have root access on the server, you then do the following, otherwise you need to ask a server admin to them for you:

1. Copy (as root/sudo) your edited copy to `/etc/init.d/` and rename it to just `evennia`: `sudo cp evennia-screen-initd.sh /etc/init.d/evennia`
2. `cd /etc/init.d`
3. Make the script owned by root: `sudo chown root:root evennia`
4. Make the script executable: `sudo chmod 755 evennia`
5. Make script run when the server starts (late in the startup): `sudo update-rc.d evennia defaults 91`

To test things work, reboot the server if you have the right to do so. When it comes back up, Evennia should be running in its (detached) Screen Session and you can connect to it as described in the previous section. If you have root access you can now also operate Evennia as a global service with `sudo service evennia start|stop|reload`.

Choosing An SQL Server

Since Evennia uses Django, most of our notes are based off of what we know from the community and their documentation. While the information below may be useful, you can always find the most up-to-date and "correct" information at Django's Notes about supported Databases page.

SQLite3

This is the default database used out of the box. SQLite stores the database in a single file (`mygame/server/evennia.db3`). This means it's very easy to reset this database - just delete (or move) that `evennia.db3` file and run `evennia migrate` again! No server process is needed and the administrative overhead and resource consumption is tiny. It is also very fast since it's run in-memory. For the vast majority of Evennia installs it will probably be all that's ever needed.

SQLite will generally be much faster than MySQL/PostGreSQL but it might not scale as well for huge databases. Its main drawback is otherwise that it does not work very well with multiple concurrent threads or processes. This has to do with file-locking clashes of the database file. So for a production server making heavy use of process- or threadpools (or when using a third-party webserver like Apache), a more full-featured database may be the better choice.

Postgres

This is Django's recommended database engine, While not as fast as SQLite for normal usage, it will scale better than SQLite, especially if your game has an very large database and/or extensive web presence through a separate server process.

Apart from the postgres server and -client for your OS, one must also `pip install psycopg2` to allow Postgresql to interface with Python. Here is a tutorial for configuring postgreSQL with Django (and thus Evennia) on Linux.

MySQL

MySQL behaves similarly to PosgreSQL. Apart from the MySQL server and -client, one must also do `pip install mysql-python` to allow MySQL to communicate with Python. Here is a tutorial for setting up MySQL with Django (and thus Evennia) on Linux.

> There is an open-source alternative to MySQL named MariaDB. While this is supposed to work the same way as MySQL, There is currently (May 2017) an apparent issue with it. Any help with verifying/resolving this is welcome!

Older versions of MySQL had some peculiarities, so check out Django's Notes about supported Databases to make sure you use the correct version.

Others

No testing has been performed with Oracle, but it is also supported through Django. There are community maintained drivers for MS SQL and possibly a few others (found via our friend, Google).

Inspecting database data

If you know SQL you can easily get command line access to your database like this:

```
evennia dbshell
```

This will drop you into the command line client for your respective database. If you don't have it you may need to install it separately; see the django dbshell documentation for more information.

> NOTE: Under Windows, in order to access SQLite dbshell you need to download the SQLite command-line shell program. It's a single executable file (sqlite3.exe) that you should place in the root of either your MUD folder or Evennia's (it's the same, in both cases Django will find it).

There are also a host of easier graphical interfaces for the various databases. For SQLite3 we recommend SQLite manager. This is a plugin for the Firefox web browser making it usable across all operating systems. Just use it to open the `mygame/server/evennia.db3` file.

Apache Config

Warning: This information is presented as a convenience, using another webserver than Evennia's own is not directly supported and you are on your own if you want to do so. Evennia's webserver works out of the box without any extra configuration and also runs in-process making sure to avoid caching race conditions. The browser web client will most likely not work (at least not without tweaking) on a third-party web server.

You can run Evennia's web front end with apache2 and mod_wsgi. However, there seems to be no reason why the codebase should not also work with other modern web servers like nginx/lighttpd + gunicorn, Tornado, uwsgi, etc.

Note that the Apache instructions below might be outdated. If something is not working right, or you use Evennia with a different server, please let us know.

mod_wsgi Setup

Install mod_wsgi

`mod_wsgi` is an excellent, secure, and high-performance way to serve Python projects. Code reloading is a breeze, Python modules are executed as a user of your choice (which is a great security win), and mod_wsgi is easy to set up on most distributions.

For the sake of brevity, this guide will refer you to mod_wsgi's installation instructions page, as their guides are great. For those that are running Debian or Ubuntu, you may install the entire stack with the following command:

```
sudo aptitude install libapache2-mod-wsgi
```

This should install apache2 (if it isn't already), `mod_wsgi`, and load the module. On Fedora or CentOS, you'll do this with `yum` and a similar package name that you'll need to search for. On Windows, you'll need to download and install apache2 and `mod_wsgi` binaries.

Copy and modify the VHOST

After `mod_wsgi` is installed, copy the `evennia/game/web/utils/evennia_wsgi_apache.conf` file to your apache2 vhosts/sites folder. On Debian/Ubuntu, this is `/etc/apache2/sites-enabled/`. Make your modifications **after** copying the file there.

Read the comments and change the paths to point to the appropriate locations within your setup.

Restart/Reload Apache

You'll then want to reload or restart apache2. On Debian/Ubuntu, this may be done via:

```
sudo /etc/init.d/apache2 restart` or `sudo /etc/init.d/apache2 reload
```

Enjoy

With any luck, you'll be able to point your browser at your domain or subdomain that you set up in your vhost and see the nifty default Evennia webpage. If not, read the hopefully informative error message and work from there. Questions may be directed to our Evennia Community site.

A note on code reloading

If your `mod_wsgi` is set up to run on daemon mode (as will be the case by default on Debian and Ubuntu), you may tell `mod_wsgi` to reload by using the `touch` command on `evennia/game/web/utils/apache_wsgi.conf`. When mod_wsgi sees that the file modification time has changed, it will force a code reload. Any modifications to the code will not be propagated to the live instance of your site until reloaded.

If you are not running in daemon mode or want to force the issue, simply restart or reload apache2 to apply your changes.

Further notes and hints:

If you get strange (and usually uninformative) `Permission denied` errors from Apache, make sure that your `evennia` directory is located in a place the webserver may actually access. For example, some Linux distributions may default to very restrictive access permissions on a user's `/home` directory.

One user commented that they had to add the following to their Apache config to get things to work. Not confirmed, but worth trying if there are trouble.

```
<Directory "/home/<yourname>/evennia/game/web">
                Options +ExecCGI
                Allow from all
</Directory>
```

Updating Your Game

Fortunately, it's extremely easy to keep your Evennia server up-to-date via GIT. If you haven't already, see the Getting Started guide and get everything running. There are many ways to get told when to update: You can subscribe to the RSS feed or manually check up on the feeds from http://www.evennia.com.

When you're wanting to apply updates, simply `cd` to your cloned `evennia` root directory and type:

```
git pull
```

assuming you've got the command line client. If you're using a graphical client, you will probably want to navigate to the `evennia` directory and either right click and find your client's pull function, or use one of the menus (if applicable).

You can review the latest changes with

```
git log
```

or the equivalent in the graphical client. You can also see the latest changes online here.

Migrating the Database Schema

Whenever we change the database layout of Evennia upstream (such as when we add new features) you will need to *migrate* your existing database. When this happens it will be clearly noted in the `git log` (it will say something to the effect of "Run migrations"). Database changes will also be announced on the Evennia mailing list.

When the database schema changes, you just go to your game folder and run

```
evennia migrate
```

Resetting your database

Should you ever want to start over completely from scratch, there is no need to re-download Evennia or anything like that. You just need to clear your database. Once you are done, you just rebuild it from scratch as described in step 2 of the Getting Started guide.

First stop a running server with

```
evennia stop
```

If you run the default `SQlite3` database (to change this you need to edit your `settings.py` file), the database is actually just a normal file in `mygame/server/` called `evennia.db3`. *Simply delete that file* - that's it. Now run `evennia migrate` to recreate a new, fresh one.

If you run some other database system you can instead flush the database:

```
evennia flush
```

This will empty the database. However, it will not reset the internal counters of the database, so you will start with higher dbref values. If this is okay, this is all you need.

Django also offers an easy way to start the database's own management should we want more direct control:

```
evennia dbshell
```

In e.g. MySQL you can then do something like this (assuming your MySQL database is named "Evennia":

```
mysql> DROP DATABASE Evennia;
mysql> exit

NOTE: Under Windows OS, in order to access SQLite dbshell you need
to `download the SQLite command-line shell program`_. It's a single
executable file (sqlite3.exe) that you should place in the root of
either your MUD folder or Evennia's (it's the same, in both cases
Django will find it).
```

More about schema migrations

If and when an Evennia update modifies the database *schema* (that is, the under-the-hood details as to how data is stored in the database), you must update your existing database correspondingly to match the change. If you don't, the updated Evennia will complain that it cannot read the database properly. Whereas schema changes should become more and more rare as Evennia matures, it may still happen from time to time.

One way one could handle this is to apply the changes manually to your database using the database's command line. This often means adding/removing new tables or fields as well as possibly convert existing data to match what the new Evennia version expects. It should be quite obvious that this quickly becomes cumbersome and error-prone. If your

database doesn't contain anything critical yet it's probably easiest to simply reset it and start over rather than to bother converting.

Enter *migrations*. Migrations keeps track of changes in the database schema and applies them automatically for you. Basically, whenever the schema changes we distribute small files called "migrations" with the source. Those tell the system exactly how to implement the change so you don't have to do so manually. When a migration has been added we will tell you so on Evennia's mailing lists and in commit messages -
you then just run `evennia migrate` to be up-to-date again.

Online Setup

Evennia development can be made also without any internet connection (except to download updates). At some point however, you are likely to want to make your game visible online, either as part of making it public or to allow other developers or beta testers access to it.

Settings for allowing external connections

By default Evennia will only allow connections from your own computer (`localhost`, the IP address `127.168.0.1`). To make it available to the outside world you need to make some changes to your settings file. To have Evennia recognize changed port settings you have to do a full stop of the server (not just a reload), via `evennia stop` before starting again.

Below is an example of a new section added to the settings. For a common setup, Evennia will require access to five computer ports, of which three (only) should be open to the outside world.

```
# in mygame/server/conf/settings.py

SERVERNAME = "MyGame"

# open to the internet: 4000, 4001, 4002
# closed to the internet (internal use): 5000, 5001
TELNET_PORTS = [4000]
WEBSOCKET_CLIENT_PORT = 4001
WEBSERVER_PORTS = [(4002, 5000)]
AMP_PORT = 5001

# security measures (optional)
TELNET_INTERFACES = ['203.0.113.0']
WEBSOCKET_CLIENT_INTERFACE = '203.0.113.0'
ALLOWED_HOSTS = [".mymudgame.com"]

# uncomment to take server offline
# LOCKDOWN_MODE = True

# Register with game index (see games.evennia.com for first setup)
GAME_DIRECTORY_LISTING = {
    'game_status': 'pre-alpha',
    'game_website': 'http://mymudgame.com:4002',
    'listing_contact': 'me@mymudgame.com',
    'telnet_hostname': 'mymudgame.com',
    'telnet_port': 4000,
    'short_description': "The official Mygame.",
```

```
    'long_description':'Mygame is ...'
}
```

When running, an external user should be able to connect with a telnet client to host `198.51.100.0` and port `4000` and use a web browser to connect to `http://203.0.113.0:4002` (in this example `http:mymudgame.com:4002` also appears to be set up to point to that IP). If connecting does not work, try turning off the `_INTERFACES` optional security settings first. If that still doesn't work, examine your firewall.

Read on for a description of the settings.

Telnet

```
# Required. Change to whichever outgoing Telnet port(s)
# you are allowed to use on your host.
TELNET_PORTS = [4000]
# Optional for security. Restrict which telnet
# interfaces we should accept. Should be set to your
# outward-facing IP address(es). Default is ´0.0.0.0´
# which accepts all interfaces.
TELNET_INTERFACES = ['0.0.0.0']
```

The `TELNET_*` settings are the most important ones for getting a traditional base game going. Which IP addresses you have available depends on your server hosting solution (see the next sections). Some hosts will restrict which ports you are allowed you use so make sure to check.

Web server

```
# Required. This is a list of tuples
# (outgoing_port, internal_port). Only the outgoing
# port should be open to the world!
# set outgoing port to 80 if you want to run Evennia
# as the only web server on your machine (if available).
WEBSERVER_PORTS = [(8000, 5001)]
# Optional for security. Change this to the IP your
# server can be reached at (normally the same
# as TELNET_INTERFACES)
WEBSERVER_INTERFACES = ['0.0.0.0']
# Optional for security. Protects against
# man-in-the-middle attacks. Change  it to your server's
# IP address or URL when you run a production server.
ALLOWED_HOSTS = ['*']
```

The web server is always configured with two ports at a time. The *outgoing* port (`8000` by default) is the port external connections can use. If you don't want users to have to specify the port when they connect, you should set this to `80` - this however only works if you are not running any other web server on the machine.
The *internal* port (`5001` by default) is used internally by Evennia to communicate between the Server and the Portal. It should not be available to the outside world. You usually only need to change the outgoing port unless the default internal port is clashing with some other program.

Web client

```
# Required. Change this to the main IP address of your server.
WEBSOCKET_CLIENT_INTERFACE = '0.0.0.0'
# Optional and needed only if using a proxy or similar. Change
# to the IP or address where the client can reach
# your server. The ws:// part is then required. If not given, the client
# will use its host location.
WEBSOCKET_CLIENT_URL = ""
# Required. Change to a free port for the websocket client to reach
# the server on. This will be automatically appended
# to WEBSOCKET_CLIENT_URL by the web client.
WEBSOCKET_CLIENT_PORT = 8001
```

The websocket-based web client needs to be able to call back to the server, and these settings must be changed for it to find where to look. If it cannot find the server you will get an warning and the client will revert to the AJAX-based (much simpler) version of the client instead.

With these changes in place, the server should be ready to accept external connections.

Other ports

```
# Required. But you should only change this if there is a clash
# with other services on your host. Should NOT be open to the
# outside world.
AMP_PORT = 5000
# Optional, only if you allow SSH connections (off by default).
SSH_PORTS = [8022]
SSH_INTERFACES = ['0.0.0.0']
# Optional, only allow SSL connections (off by default).
SSL_PORTS = [4001]
SSL_INTERFACES = ['0.0.0.0']
```

The `AMP_PORT` is required to work, since this is the internal port linking Evennia's Server and Portal components together. The other ports are encrypted ports that may be useful for custom protocols but are otherwise not used.

Lockdown mode

When you test things out and check configurations you may not want players to drop in on you. Similarly, if you are doing maintenance on a live game you may want to take it offline for a while to fix eventual problems without risking people connecting. To do this, stop the server with `evennia stop` and add `LOCKDOWN_MODE = True` to your settings file. When you start the server again, your game will only be accessible from localhost.

Registering with the Evennia game directory

Once your game is online you should make sure to register it with the Evennia Game Index. Registering with the index will help drum up interest for your game and also shows people that Evennia is being used. It's a convenient place to reference your game from (if you don't have a fancy URL set up yet). You *don't* need to have a fully-fledged production game to go on the game index. If it's very rough just set the development status to "pre-alpha".

In the example settings snippet above is an example registry entry you can modify for your game. To activate the directory client you also need to add three lines to `mygame/server/conf/server_services_plugins.py`:

```
# in mygame/server/conf/server_services_plugins.py`

from evennia.contrib.egi_client import EvenniaGameIndexService  # ADD

def start_plugin_services(server):
    """
    This hook is called by Evennia, last in the Server startup process.

    server - a reference to the main server application.
    """
    egi_service = EvenniaGameIndexService()  # ADD
    server.services.addService(egi_service)  # ADD
```

Once that's in place, just restart your game. See the EGI client documentation for more info.

Using your own computer as a server

By far the simplest and probably cheapest option. Evennia will run on your own home computer. Moreover, since Evennia is its own web server, you don't need to install anything extra to have a website.

Advantages

- Free (except for internet costs and the electrical bill).
- Full control over the server and hardware (it sits right there!).
- Easy to set up.
- Also suitable for quick setups - e.g. to briefly show off results to your collaborators.

Disadvantages

- You need a good internet connection, ideally without any upload/download limits/costs.
- If you want to run a full game this way, your computer needs to always be on. It could be noisy, and as mentioned, the electrical bill must be considered.
- No support or safety - if your house burns down, so will your game. Also, you are yourself responsible for doing regular backups.
- Potentially not as easy if you don't know how open ports in your firewall or router.
- Home IP numbers are often dynamically allocated, so for permanent online time you need to set up a DNS to always re-point to the right place (see below).

Setting up your own machine as a server

Making Evennia available from your own machine is mainly a matter of configuring eventual firewalls to let Evennia's communication through. With Evennia running, note which outgoing ports it is using. Default ports are 4000 for telnet, 8000 for the website and port 8001 for the websocket-based client. Note that if the websocket port is not open, Evennia will fall back to the website port and use a less able version of the client. We assume the defaults below.

1. Go to http://www.whatismyip.com/ (or similar site). They should tell you which IP address you are connecting from, let's say it is `230.450.0.222`.
2. In your web browser, go to `http://230.450.0.222:8000`, where the last `:8000` is the webclient port Evennia uses. If you see Evennia's website and can connect to the webclient - -congrats, you are almost there! Click on the webclient link. Next try to connect with a traditional MUD-client to the telnet port.

3. Most likely your ports won't work straight off though. This is probably because you have a firewall blocking the ports we need. How to open the right ports in your software firewall is something you need to look up in the manual for your firewall. There could also be a hardware-router between your computer and the Internet - in that case the IP address we see "from the outside" is actually the router's IP, not that of your computer on your local network. The router has ports of its own that need to be opened.

- If your computer's software firewall is blocking Evennia's outgoing ports, you need to open those ports in your firewall software. Please refer to the manual/helpfile for your firewall on how to do this.
- If you are (maybe in addition to a software firewall) using a hardware router/firewall, this has its own internet-facing IP and ports. Exactly how to configure it is again depending on brand and version. In principle you should look for something called "Port forwarding" or maybe "Virtual server". You want to route Evennia's outgoing ports 4000/8000/8001 from your computer to equivalent "router outgoing ports" that the world can see. The router's outgoing ports do *not* have to have to have the same port numbers as evennia's outgoing ports! For example, you might want to connect evennia's outgoing port 8000 to an outgoing router port 80 - this is the port HTTP requests use and web browsers automatically look for. If you use port 80 you won't have to specify the port number in the url of your browser. This would collide with any other web services you are running through this router though.
- For some hardware routers you have to reboot the router for the firewall changes to to take effect. It's worth a try if your changes doesn't seem to come into effect.

1. At this point you should be able to invite people to play your game on `http://230.450.0.222:8000` or via telnet to `230.450.0.222` on port `4000`.

A complication with using a specific IP address like this is that your home IP might not remain the same. Many ISPs (Internet Service Providers) allocates a dynamic IP to you which could change at any time. When that happens, that IP you told people to go to will be worthless. Also, that long string of numbers is not very pretty, is it? It's hard to remember and not easy to use in marketing your game. What you need is to alias it to a more sensible domain name - an alias that follows you around also when the IP changes.

1. To set up a domain name alias, we recommend starting with a free domain name from FreeDNS. Once you register there (it's free) you have access to tens of thousands domain names that people have "donated" to allow you to use for your own sub domain. For example, `strangled.net` is one of those available domains. So tying our IP address to `strangled.net` using the subdomain `evennia` would mean that one could henceforth direct people to `http://evennia.strangled.net:8000` for their gaming needs - far easier to remember!
2. So how do we make this new, nice domain name follow us also if our IP changes? For this we need to set up a little program on our computer. It will check whenever our ISP decides to change our IP and tell FreeDNS that. There are many alternatives to be found from FreeDNS:s homepage, one that works on multiple platforms is inadyn. Get it from their page or, in Linux, through something like `apt-get install inadyn`.
3. Next, you login to your account on FreeDNS and go to the Dynamic page. You should have a list of your subdomains. Click the `Direct URL` link and you'll get a page with a text message. Ignore that and look at the URL of the page. It should be ending in a lot of random letters. Everything after the question mark is your unique "hash". Copy this string.
4. You now start inadyn with the following command (Linux):

```
inadyn --dyndns_system default@freedns.afraid.org -a <my.domain>,<hash> &
```

where `<my.domain>` would be `evennia.strangled.net` and `<hash>` the string of numbers we copied from FreeDNS. The `&` means we run in the background (might not be valid in other operating systems). `inadyn` will henceforth check for changes every 60 seconds. You should put the `inadyn` command string in a startup script somewhere so it kicks into gear whenever your computer starts.

Remote hosting

Your normal "web hotel" will probably not be enough to run Evennia. A web hotel is normally aimed at a very specific usage - delivering web pages, at the most with some dynamic content. The "Python scripts" they refer to on their home pages are usually only intended to be CGI-like scripts launched by their webserver. Even if they allow you shell access (so you can install the Evennia dependencies in the first place), resource usage will likely be very restricted. Running a full-fledged game server like Evennia will probably be shunned upon or be outright impossible. If you are unsure, contact your web hotel and ask about their policy on you running third-party servers that will want to open custom ports.

The options you probably need to look for are *shell account services*, *VPS:es* or *Cloud services*. A "Shell account" service means that you get a shell account on a server and can log in like any normal user. By contrast, a *VPS* (Virtual Private Server) service usually means that you get `root` access, but in a virtual machine. There are also *Cloud*-type services which allows for starting up multiple virtual machines and pay for what resources you use.

Advantages

- Shell accounts/VPS/clouds offer more flexibility than your average web hotel - it's the ability to log onto a shared computer away from home.
- Usually runs a Linux flavor, making it easy to install Evennia.
- Support. You don't need to maintain the server hardware. If your house burns down, at least your game stays online. Many services guarantee a certain level of up-time and also do regular backups for you. Make sure to check, some offer lower rates in exchange for you yourself being fully responsible for your data/backups.
- Usually offers a fixed domain name, so no need to mess with IP addresses.

Disadvantages

- Might be pretty expensive (more so than a web hotel). Note that Evennia will normally need at least 50MB RAM and likely much more for a large production game.
- Linux flavors might feel unfamiliar to users not used to ssh/PuTTy and the Linux command line.
- You are probably sharing the server with many others, so you are not completely in charge. CPU usage might be limited. Also, if the server people decides to take the server down for maintenance, you have no choice but to sit it out (but you'll hopefully be warned ahead of time).

Installing Evennia on a remote server

Assuming you know how to connect to your account over ssh/PuTTy you should be able to follow the Getting Started instructions normally. You only need Python and GIT pre-installed; these should both be available on any servers (if not you should be able to easily ask for them to be installed). On a VPS or Cloud service you can install them yourself as needed.

If `virtualenv` is not available and you can't get it, you can download it (it's just a single file) from the virtualenv pypi. Using `virtualenv` you can install everything without actually needing to have further `root` access. Ports might be an issue, so make sure you know which ports are available to use.

Hosting options

To find commercial solutions, browse the web for "shell access", "VPS" or "Cloud services" in your region. You may find useful offers for "low cost" VPS hosting on Low End Box. The associated Low End Talk forum can be useful for health checking the many small businesses that offer "value" hosting, and occasionally for technical suggestions.

There are all sorts of services available. Below are some international suggestions offered by Evennia users:

Hosting name | Type | Lowest price | Comments
———————-|:———————:|:———-:|———————-
silvren.com | Shell account | Free for MU* | Private hobby provider so don't assume backups or expect immediate support. To ask for an account, connect with a MUD client to iweb.localecho.net, port 4201 and ask for "Jarin".
Digital Ocean | VPS | $5/month | Please consider using the referral link https://m.do.co/c/4044f187216f. You will get a $10 credit - and once you have paid $25 we will get that as a referral bonus to help Evennia development.
Amazon Web services | Cloud | ~$5/month / on-demand | Free Tier first 12 months. Regions available around the globe.
Amazon Lightsail | Cloud | $5/month | Free first month. AWS's new "fixed cost" offering.
Genesis MUD hosting | Shell account | $8/month | Dedicated MUD host with very limited memory offerings. Note that Evennia needs *at least* the "Deluxe" package (50MB RAM) and probably *a lot* higher for a production game, so be very careful about choosing this host for Evennia.
Host1Plus | VPS & Cloud | $4/month | $4-$8/month depending on length of sign-up period.
Scaleway | Cloud | €3/month / on-demand | EU based (Paris, Amsterdam). Smallest option provides 2GB RAM.
Prgmr | VPS | $5/month | 1 month free with a year prepay. You likely want some experience with servers with this option as they don't have a lot of support.
Linode | Cloud | $5/month / on-demand | Multiple regions. Smallest option provides 1GB RAM
Please help us expand this list.

Cloud9

If you are interested in running Evennia in the online dev environment Cloud9, you can spin it up through their normal online setup using the Evennia Linux install instructions. The one extra thing you will have to do is update `mygame/server/conf/settings.py` and add `WEBSERVER_PORTS = [(8080, 5001)]`. This will then let you access the web server and do everything else as normal.

SSL

SSL can be very useful for web clients. It will protect the credentials and gameplay of your users over a web client if they are in a public place, and your websocket can also be switched to WSS for the same benefit. SSL certificates used to cost money on a yearly basis, but there is now a program that issues them for free with assisted setup to make the entire process less painful.

Let's Encrypt

Let's Encrypt is a certificate authority offering free certificates to secure a website with HTTPS. To get started issuing a certificate for your web server using Let's Encrypt, see these links:

- Let's Encrypt - Getting Started
- The CertBot Client is a program for automatically obtaining a certificate, use it and maintain it with your website.

Also, on Freenode visit the #letsencrypt channel for assistance from the community. For an additional resource, Let's Encrypt has a very active community forum.

A blog where someone sets up Let's Encrypt

The only process missing from all of the above documentation is how to pass verification. This is how Let's Encrypt verifies that you have control over your domain (not necessarily ownership, it's Domain Validation (DV)). This can be done either with configuring a certain path on your web server or through a TXT record in your DNS. Which one you will want to do is a personal preference, but can also be based on your hosting choice. In a controlled/cPanel environment, you will most likely have to use DNS verification.

CHAPTER 8

Admin Documentation

This chapter is aimed at game administrators – the higher-ups that possess shell access and are responsible for managing the game.

Banning

Whether due to abuse, blatant breaking of your rules, or some other reason you will eventually find no other recourse but to kick out a particularly troublesome player. The default command set has admin tools to handle this, primarily `@ban`, `@unban` and `@boot`.

Creating a ban

Say we have a troublesome player "YouSuck" - this is a guy that refuse common courtesy - an abusive and spammy account that is clearly created by some bored internet hooligan only to cause grief. You have tried to be nice. Now you just want this troll gone.

Name ban

The easiest is to block the account YouSuck from ever connecting again.

```
@ban YouSuck
```

This will lock the name YouSuck (as well as 'yousuck' and any other combination), and next time they try to log in with this name the server will not let them!

You can also give a reason so you remember later why this was a good thing (the banned player will never see this)

```
@ban YouSuck:This is just a troll.
```

If you are sure this is just a spam account, you might even consider deleting the player account outright:

```
@delplayer YouSuck
```

Generally banning the name is the easier and safer way to stop the use of an account – if you change your mind you can always remove the block later whereas a deletion is permanent.

IP ban

Just because you block YouSuck's name might not mean the trolling human behind that account gives up. They can just create a new account YouSuckMore and be back at it. One way to make things harder for them is to tell the server to not allow connections from their particular IP address.

First, when the offending player is online, check which IP address they use. This you can do with the `who` command, which will show you something like this:

```
Player Name      On for      Idle      Room      Cmds      Host
YouSuck          01:12       2m        22        212       237.333.0.223
```

The "Host" bit is the IP address from which the player is connecting. Use this to define the ban instead of the name:

```
@ban 237.333.0.223
```

This will stop YouSuck connecting from his computer. Note however that IP addresses might change easily - either due to how the player's Internet Service Provider operates or by the user simply changing computer. You can make a more general ban by putting asterisks `*` as wildcards for the groups of three digits in the address. So if you figure out that !YouSuck mainly connects from 237.333.0.223, 237.333.0.225 and 237.333.0.256 (only changes in the local subnet), it might be an idea to put down a ban like this to include any number in that subnet:

```
@ban 237.333.0.*
```

You should combine the IP ban with a name-ban too of course, so the account YouSuck is truly locked regardless of from where they connect.

Be careful with too general IP bans however (more asterisks above). If you are unlucky you could be blocking out innocent players who just happen to connect from the same subnet as the offender.

Booting

YouSuck is not really noticing all this banning yet though - and won't until having logged out and tries to log back in again. Let's help the troll along.

```
@boot YouSuck
```

Good riddance. You can give a reason for booting too (to be echoed to the player before getting kicked out).

```
@boot YouSuck:Go troll somewhere else.
```

Lifting a ban

Give the `@unban` (or `@ban`) command without any arguments and you will see a list of all currently active bans:

```
Active bans
id    name/ip       date                        reason
1     yousuck       Fri Jan 3 23:00:22 2020     This is just a Troll.
2     237.333.0.*   Fri Jan 3 23:01:03 2020     YouSuck's IP.
```

Use the `id` from this list to find out which ban to lift.

```
 @unban 2

Cleared ban 2: 237.333.0.*
```

IRC

IRC (Internet Relay Chat) is a long standing chat protocol used by many open-source projects for communicating in real time. By connecting one of Evennia's Channels to an IRC channel you can communicate also with people not on an mud themselves. You can also use IRC if you are only running your Evennia MUD locally on your computer (your game doesn't need to be open to the public)! All you need is an internet connection. For IRC operation you also need twisted.words. This is available simply as a package *python-twisted-words* in many Linux distros, or directly downloadable from the link.

Configuring IRC

To configure IRC, you'll need to activate it in your settings file.

```
IRC_ENABLED = True
```

Start Evennia and log in as a privileged user. You should now have a new command available: `@irc2chan`. This command is called like this:

```
@irc2chan[/switches] <evennia_channel> = <ircnetwork> <port> <#irchannel> <botname>
```

If you already know how IRC works, this should be pretty self-evident to use. Read the help entry for more features.

Setting up IRC, step by step

You can connect IRC to any Evennia channel (so you could connect it to the default *public* channel if you like), but for testing, let's set up a new channel `irc`.

```
@ccreate irc = This is connected to an irc channel!
```

You will automatically join the new channel.

Next we will create a connection to an external IRC network and channel. There are many, many IRC nets. Here is a list of some of the biggest ones, the one you choose is not really very important unless you want to connect to a particular channel (also make sure that the network allows for "bots" to connect).

For testing, we choose the *Freenode* network, `irc.freenode.net`. We will connect to a test channel, let's call it *#myevennia-test* (an IRC channel always begins with `#`). It's best if you pick an obscure channel name that didn't exist previously - if it didn't exist it will be created for you.

> *Don't* connect to `#evennia` for testing and debugging, that is Evennia's official chat channel! You *are* welcome to connect your game to `#evennia` once you have everything working though - it can be a good way to get help and ideas. But if you do, please do so with an in-game channel open only to your game admins and developers).

The *port* needed depends on the network. For Freenode this is `6667`.

What will happen is that your Evennia server will connect to this IRC channel as a normal user. This "user" (or "bot") needs a name, which you must also supply. Let's call it "mud-bot".

To test that the bot connects correctly you also want to log onto this channel with a separate, third-party IRC client. There are hundreds of such clients available. If you use Firefox, the *Chatzilla* plugin is good and easy. Freenode also offers its own web-based chat page. Once you have connected to a network, the command to join is usually `/join #channelname` (don't forget the #).

Next we connect Evennia with the IRC channel.

```
@irc2chan irc = irc.freenode.net 6667 #myevennia-test mud-bot
```

Evennia will now create a new IRC bot `mud-bot` and connect it to the IRC network and the channel #myevennia. If you are connected to the IRC channel you will soon see the user *mud-bot* connect.

Write something in the Evennia channel *irc*.

```
 irc Hello, World!
[irc] Anna: Hello, World!
```

If you are viewing your IRC channel with a separate IRC client you should see your text appearing there, spoken by the bot:

```
mud-bot> [irc] Anna: Hello, World!
```

Write `Hello!` in your IRC client window and it will appear in your normal channel, marked with the name of the IRC channel you used (#evennia here).

```
[irc] Anna@#myevennia-test: Hello!
```

Your Evennia gamers can now chat with users on external IRC channels!

RSS

RSS is a format for easily tracking updates on websites. The principle is simple - whenever a site is updated, a small text file is updated. An RSS reader can then regularly go online, check this file for updates and let the user know what's new.

Evennia allows for connecting any number of RSS feeds to any number of in-game channels. Updates to the feed will be conveniently echoed to the channel. There are many potential uses for this: For example the MUD might use a separate website to host its forums. Through RSS, the players can then be notified when new posts are made. Another example is to let everyone know you updated your dev blog. Admins might also want to track the latest Evennia updates through our own RSS feed here.

Configuring RSS

To use RSS, you first need to install the feedparser python module.

```
pip install feedparser
```

Next you activate RSS support in your config file by settting `RSS_ENABLED=True`.

Start/reload Evennia as a privileged user. You should now have a new command available, `@rss2chan`:

```
@rss2chan <evennia_channel> = <rss_url>
```

Setting up RSS, step by step

You can connect RSS to any Evennia channel, but for testing, let's set up a new channel "rss".

```
@ccreate rss = RSS feeds are echoed to this channel!
```

Let's connect Evennia's code-update feed to this channel. The RSS url for evennia updates is `https://github.com/evennia/evennia/commits/master.atom`, so let's add that:

```
@rss2chan rss = https://github.com/evennia/evennia/commits/master.atom
```

That's it, really. New Evennia updates will now show up as a one-line title and link in the channel. Give the `@rss2chan` command on its own to show all connections. To remove a feed from a channel, you specify the connection again (use the command to see it in the list) but add the `/delete` switch:

```
@rss2chan/delete rss = https://github.com/evennia/evennia/commits/master.atom
```

You can connect any number of RSS feeds to a channel this way. You could also connect them to the same channels as IRC to have the feed echo to external chat channels as well.

Text Encodings

Evennia is a text-based game server. This makes it important to understand how it actually deals with data in the form of text.

Text *byte encodings* describe how a string of text is actually stored in the computer - that is, the particular sequence of bytes used to represent the letters of your particular alphabet. A common encoding used in English-speaking languages is the *ASCII* encoding. This describes the letters in the English alphabet (Aa-Zz) as well as a bunch of special characters. For describing other character sets (such as that of other languages with other letters than English), sets with names such as *Latin-1*, *ISO-8859-3* and *ARMSCII-8* are used. There are hundreds of different byte encodings in use around the world.

In contrast to the byte encoding is the *unicode representation*. The unicode is an internationally agreed-upon table describing essentially all available letters you could ever want to print. Everything from English to Chinese alphabets and all in between. So what Evennia (as well as Python and Django) does is to store everything in Unicode internally, but then converts the data to one of the encodings whenever outputting data to the user.

The problem is that when receiving a string of bytes from a user it's impossible for Evennia to guess which encoding was used - it's just a bunch of bytes! Evennia must know the encoding in order to convert back and from the correct unicode representation.

How to customize encodings

As long as you stick to the standard ASCII character set (which means the normal English characters, basically) you should not have to worry much about this section.

If you want to build your game in another language however, or expect your users to want to use special characters not in ASCII, you need to consider which encodings you want to support.

As mentioned, there are many, many byte-encodings used around the world. It should be clear at this point that Evennia can't guess but has to assume or somehow be told which encoding you want to use to communicate with the server. Basically the encoding used by your client must be the same encoding used by the server. This can be customized in two complementary ways.

1. Point users to the default `@encoding` command or the `@options` command. This allows them to themselves set which encoding they (and their client of choice) uses. Whereas data will remain stored as unicode internally in Evennia, all data received from and sent to this particular player will be converted to the given format before transmitting.
2. As a back-up, in case the user-set encoding translation is erroneous or fails in some other way, Evennia will fall back to trying with the names defined in the settings variable `ENCODINGS`. This is a list of encoding names Evennia will try, in order, before giving up and giving an encoding error message.

Note that having to try several different encodings every input/output adds unneccesary overhead. Try to guess the most common encodings you players will use and make sure these are tried first. The International *UTF-8* encoding is what Evennia assumes by default (and also what Python/Django use normally). See the Wikipedia article here for more help.

Internationalization

Internationalization (often abbreviated *i18n* since there are 18 characters between the first "i" and the last "n" in that word) allows Evennia's core server to return texts in other languages than English - without anyone having to edit the source code. Take a look at the `locale` directory of the Evennia installation, there you will find which languages are currently supported.

Changing server language

Change language by adding the following to your `mygame/server/conf/settings.py` file:

```
USE_I18N = True
LANGUAGE_CODE = 'en'
```

Here `'en'` should be changed to the abbreviation for one of the supported languages found in `locale/`. Restart the server to activate i18n. The two-character international language codes are found here.

> Windows Note: If you get errors concerning `gettext` or `xgettext` on Windows, see the Django documentation. A self-installing and up-to-date version of gettext for Windows (32/64-bit) is available on Github.

Translating Evennia

> **Important Note:** Evennia offers translations of hard-coded strings in the server, things like "Connection closed" or "Server restarted", strings that end users will see and which game devs are not supposed to change on their own. Text you see in the log file or on the command line (like error messages) are generally *not* translated (this is a part of Python).
>
> In addition, text in default Commands and in default Typeclasses will *not* be translated by switching *i18n* language. To translate Commands and Typeclass hooks you must overload them in your game directory and translate their returns to the language you want. This is because from Evennia's perspective, adding *i18n* code to commands tend to add complexity to code that is *meant* to be changed anyway. One of the goals of Evennia is to keep the user-changeable code as clean and easy-to-read as possible.

If you cannot find your language in `evennia/locale/` it's because noone has translated it yet. Alternatively you might have the language but find the translation bad ... You are welcome to help improve the situation!

To start a new translation you need to first have cloned the Evennia repositry with GIT and activated a python virtualenv as described on the Getting Started page. You now need to `cd` to the `evennia/` directory. This is *not* your created game folder but the main Evennia library folder. If you see a folder `locale/` then you are in the right place. From here you run:

```
django-admin makemessages -l <language-code>
```

where `<language-code>` is the two-letter locale code for the language you want, like 'sv' for Swedish or 'es' for Spanish. After a moment it will tell you the language has been processed. If you started a new language a new folder for that language will have emerged in the `locale/` folder. Otherwise the system will just have updated the existing translation with eventual new strings found in the server. Running this command will not overwrite any existing strings so you can run it as much as you want.

Next head to `locale/<language-code>/LC_MESSAGES` and edit the `**.po` file you find there. You can edit this with a normal text editor but it is easiest if you use a special po-file editor from the web (search the web for "po editor" for many free alternatives).

The concept of translating is simple, it's just a matter of taking the english strings you find in the `**.po` file and add your language's translation best you can. The `**.po` format (and many supporting editors) allow you to mark translations as "fuzzy". This tells the system (and future translators) that you are unsure about the translation, or that you couldn't find a translation that exactly matched the intention of the original text. Other translators will see this and might be able to improve it later.

Finally, you need to compile your translation into a more efficient form. Do so from the `evennia` folder again:

```
django-admin compilemessages
```

This will go through all languages and create/update compiled files (`**.mo`) for them. This needs to be done whenever a `**.po` file is updated.

When you are done, send the `**.po` and `*.mo` file to the Evennia developer list (or push it into your own repository clone) so we can integrate your translation into Evennia!

CHAPTER 9

Builder Documentation

This chapter contains information useful to world builders and designers.

Building Quickstart

The default command definitions coming with Evennia follows a style similar to that of MUX, so the commands should be familiar if you used any such code bases before. If you haven't, you might be confused by the use of `@` in places. This is just a naming convention - commands related to out-of-character or admin-related actions tend to start with `@`, the symbol has no meaning of its own.

The default commands have the following style (where `[...]` marks optional parts):

```
command[/switch/switch...] [arguments ...]
```

A *switch* is a special, optional flag to the command to make it behave differently. It is always put directly after the command name, and begins with a forward slash (`/`). The *arguments* are one or more inputs to the commands. It's common to use an equal sign (=) when assigning something to an object.

Below are some examples of commands you can try when logged in to the game. Use `help <command>` for learning more about each command and their detailed options.

Stepping down from godhood

If you just installed Evennia, your very first player account is called user #1, also known as the *superuser* or *god user*. This user is very powerful, so powerful that it will override many game restrictions such as locks. This can be useful, but it also hides some functionality that you might want to test.

To temporarily step down from your superuser position you can use the `@quell` command in-game:

```
@quell
```

This will make you start using the permission of your current character's level instead of your superuser level. If you didn't change any settings your game Character should have an *Immortal* level permission - high as can be without

bypassing locks like the superuser does. This will work fine for the examples on this page. Use `@unquell` to get back to superuser status again afterwards.

Creating an object

Basic objects can be anything – swords, flowers and non-player characters. They are created using the `@create` command:

```
@create box
```

This created a new 'box' (of the default object type) in your inventory. Use the command `inventory` (or `i`) to see it. Now, 'box' is a rather short name, let's rename it and tack on a few aliases.

```
@name box = very large box;box;very;crate
```

We now renamed the box to *very large box* (and this is what we will see when looking at it), but we will also recognize it by any of the other names we give - like *crate* or simply *box* as before. We could have given these aliases directly after the name in the `@create` command, this is true for all creation commands - you can always tag on a list of `;`-separated aliases to the name of your new object. If you had wanted to not change the name itself, but to only add aliases, you could have used the `@alias` command.

We are currently carrying the box. Let's drop it (there is also a short cut to create and drop in one go by using the `/drop` switch, for example `@create/drop box`).

```
drop box
```

Hey presto - there it is on the ground, in all its normality.

```
examine box
```

This will show some technical details about the box object. For now we will ignore what this information means.

Try to `look` at the box to see the (default) description.

```
look box
```

The description you get is not very exciting. Let's add some flavor.

```
@describe box = This is a large and very heavy box.
```

If you try the `get` command we will pick up the box. So far so good, but if we really want this to be a large and heavy box, people should *not* be able to run off with it that easily. To prevent this we need to lock it down. This is done by assigning a *Lock* to it. Make sure the box was dropped in the room, then try this:

```
@lock box = get:false()
```

Locks represent a rather big topic, but for now that will do what we want. This will lock the box so noone can lift it. The exception is superusers, they override all locks and will pick it up anyway. Make sure you are quelling your superuser powers and try to get the box now:

```
> get box
You can't get that
```

Think thís default error message looks dull? The `get` command looks for an Attribute named `get_err_msg` for returning a nicer error message (we just happen to know this, you would need to peek into the code for the `get` command to find out.). You set attributes using the `@set` command:

```
@set box/get_err_msg = It's way too heavy for you to lift.
```

Try to get it now and you should see a nicer error message echoed back to you.

You create new Commands (or modify existing ones) in Python outside the game. See the Adding Commands tutorial for help with creating your first own Command.

Get a personality

Scripts are powerful things that allows time-dependent effects on objects. To try out a first script, let's put one on ourselves. There is an example script in `evennia/contrib/tutorial_examples/bodyfunctions.py` that is called `BodyFunctions`. To add this to us we will use the `@script` command:

```
@script self = tutorial_examples.bodyfunctions.BodyFunctions
```

(note that you don't have to give the full path as long as you are pointing to a place inside the `contrib` directory, it's one of the places Evennia looks for Scripts). Wait a while and you will notice yourself starting making random observations.

```
@script self
```

This will show details about scripts on yourself (also `examine` works). You will see how long it is until it "fires" next. Don't be alarmed if nothing happens when the countdown reaches zero - this particular script has a randomizer to determine if it will say something or not. So you will not see output every time it fires.

When you are tired of your character's "insights", kill the script with

```
@script/stop self = tutorial_examples.bodyfunctions.BodyFunctions
```

You create your own scripts in Python, outside the game; the path you give to `@script` is literally the Python path to your script file. The Scripts page explains more details.

Pushing your buttons

If we get back to the box we made, there is only so much fun you can do with it at this point. It's just a dumb generic object. If you renamed it to `stone` and changed its description noone would be the wiser. However, with the combined use of custom Typeclasses, Scripts and object-based Commands, you could expand it and other items to be as unique, complex and interactive as you want.

Let's take an example. So far we have only created objects that use the default object typeclass named simply `Object`. Let's create an object that is a little more interesting. Under `evennia/contrib/tutorial_examples` there is a module `red_button.py`. It contains the enigmatic `RedButton` typeclass.

Let's make us one of *those*!

```
@create/drop button:tutorial_examples.red_button.RedButton
```

We import the RedButton python class the same way you would import it in Python except Evennia makes sure to look in`evennia/contrib/` so you don't have to write the full path every time. There you go - one red button.

The RedButton is an example object intended to show off a few of Evennia's features. You will find that the Typeclass and Commands controlling it inside `evennia/contrib/tutorial_examples/`.

If you wait for a while (make sure you dropped it!) the button will blink invitingly. Why don't you try to push it . . . ? Surely a big red button is meant to be pushed. You know you want to.

Making yourself a house

The main command for shaping the game world is `@dig`. For example, if you are standing in Limbo you can dig a route to your new house location like this:

```
@dig house = large red door;door;in,to the outside;out
```

This will create a new room named 'house'. Spaces at the start/end of names and aliases are ignored so you could put more air if you wanted. This call will directly create an exit from your current location named 'large red door' and a corresponding exit named 'to the outside' in the house room leading back to Limbo. We also define a few aliases to those exits, so people don't have to write the full thing all the time.

If you wanted to use normal compass directions (north, west, southwest etc), you could do that with `@dig` too. But Evennia also has a limited version of `@dig` that helps for compass directions (and also up/down and in/out). It's called `@tunnel`:

```
@tunnel sw = cliff
```

This will create a new room "cliff" with an exit "southwest" leading there and a path "northeast" leading back from the cliff to your current location.

You can create new exits from where you are using the `@open` command:

```
@open north;n = house
```

This opens an exit `north` (with an alias `n`) to the previously created room `house`.

If you have many rooms named `house` you will get a list of matches and have to select which one you want to link to. You can also give its database (#dbref) number, which is unique to every object. This can be found with the `examine` command or by looking at the latest constructions with `@objects`.

Follow the north exit to your 'house' or `@teleport` to it:

```
north
```

or:

```
@teleport house
```

To manually open an exit back to Limbo (if you didn't do so with the `@dig` command):

```
@open door = limbo
```

(or give limbo's dbref which is #2)

Reshuffling the world

You can find things using the `@find` command. Assuming you are back at `Limbo`, let's teleport the *large box to our house*.

```
> @teleport box = house
very large box is leaving Limbo, heading for house.
Teleported very large box -> house.
```

We can still find the box by using @find:

```
> @find box
One Match(#1-#8):
very large box(#8) - src.objects.objects.Object
```

Knowing the `#dbref` of the box (#8 in this example), you can grab the box and get it back here without actually yourself going to `house` first:

```
@teleport #8 = here
```

(You can usually use `here` to refer to your current location. To refer to yourself you can use `self` or `me`). The box should now be back in Limbo with you.

We are getting tired of the box. Let's destroy it.

```
@destroy box
```

You can destroy many objects in one go by giving a comma-separated list of objects (or their #dbrefs, if they are not in the same location) to the command.

Adding a help entry

An important part of building is keeping the help files updated. You can add, delete and append to existing help entries using the `@sethelp` command.

```
@sethelp/add MyTopic = This help topic is about ...
```

Adding a World

After this brief introduction to building you may be ready to see a more fleshed-out example. Evennia comes with a tutorial world for you to explore.

First you need to switch back to *superuser* by using the `@unquell` command. Next, place yourself in `Limbo` and run the following command:

```
@batchcommand tutorial_world.build
```

This will take a while (be patient and don't re-run the command). You will see all the commands used to build the world scroll by as the world is built for you.

You will end up with a new exit from Limbo named *tutorial*. Apart from being a little solo-adventure in its own right, the tutorial world is a good source for learning Evennia building (and coding).

Read the batch file to see exactly how it's built, step by step. See also more info about the tutorial world here.

Connection Screen

When you first connect to your game you are greeted by Evennia's default connection screen.

```
============================================================
 Welcome to Evennia, version Beta-ra4d24e8a3cab+!

 If you have an existing account, connect to it by typing:
      connect <username> <password>
```

```
 If you need to create an account, type (without the <>'s):
      create <username> <password>

 If you have spaces in your username, enclose it in quotes.
 Enter help for more info. look will re-show this screen.
========================================================
```

Effective, but not very exciting. You will most likely want to change this to be more unique for your game. This is simple:

1. Edit `mygame/server/conf/connection_screens.py`.
2. Reload Evennia.

Evennia will look into this module and locate all *globally defined strings* in it. These strings are used as the text in your connection screen and are shown to the user at startup. If more than one such string/screen is defined in the module, a *random* screen will be picked from among those available.

Commands available at the Connection Screen

You can also customize the Commands available to use while the connection screen is shown (`connect`, `create` etc). These commands are a bit special since when the screen is running the player is not yet logged in. A command is made available at the login screen by adding them to `UnloggedinCmdSet` in `mygame/commands/default_cmdset.py`. See Commands and the tutorial section on how to add new commands to a default command set.

Batch Processors

Building a game world is a lot of work, especially when starting out. Rooms should be created, descriptions have to be written, objects must be detailed and placed in their proper places. In many traditional MUD setups you had to do all this online, line by line, over a telnet session.

Evennia already moves away from much of this by shifting the main coding work to external Python modules. But also building would be helped if one could do some or all of it externally. Enter Evennia's *batch processors* (there are two of them). The processors allows you, as a game admin, to build your game completely offline in normal text files (*batch files*) that the processors understands. Then, when you are ready, you use the processors to read it all into Evennia (and into the database) in one go.

You can of course still build completely online should you want to - this is certainly the easiest way to go when learning and for small build projects. But for major building work, the advantages of using the batch-processors are many:

- It's hard to compete with the comfort of a modern desktop text editor; Compared to a traditional MUD line input, you can get much better overview and many more features. Also, accidentally pressing Return won't immediately commit things to the database.
- You might run external spell checkers on your batch files. In the case of one of the batch-processors (the one that deals with Python code), you could also run external debuggers and code analyzers on your file to catch problems before feeding it to Evennia.
- The batch files (as long as you keep them) are records of your work. They make a natural starting point for quickly re-building your world should you ever decide to start over.
- If you are an Evennia developer, using a batch file is a fast way to setup a test-game after having reset the database.
- The batch files might come in useful should you ever decide to distribute all or part of your world to others.

There are two batch processors, the Batch-*command* processor and the Batch-*code* processor. The first one is the simpler of the two. It doesn't require any programming knowledge - you basically just list in-game commands in a text file. The code-processor on the other hand is much more powerful but also more complex - it lets you use Evennia's API to code your world in full-fledged Python code.

- The Batch Command Processor
- The Batch Code Processor

If you plan to use international characters in your batchfiles you are wise to read about *file encodings* below.

A note on File Encodings

As mentioned, both the processors take text files as input and then proceed to process them. As long as you stick to the standard ASCII character set (which means the normal English characters, basically) you should not have to worry much about this section.

Many languages however use characters outside the simple `ASCII` table. Common examples are various apostrophes and umlauts but also completely different symbols like those of the greek or cyrillic alphabets.

First, we should make it clear that Evennia itself handles international characters just fine. It (and Django) uses unicode strings internally.

The problem is that when reading a text file like the batchfile, we need to know how to decode the byte-data stored therein to universal unicode. That means we need an *encoding* (a mapping) for how the file stores its data. There are many, many byte-encodings used around the world, with opaque names such as `Latin-1`, `ISO-8859-3` or `ARMSCII-8` to pick just a few examples. Problem is that it's practially impossible to determine which encoding was used to save a file just by looking at it (it's just a bunch of bytes!). You have to *know*.

With this little introduction it should be clear that Evennia can't guess but has to *assume* an encoding when trying to load a batchfile. The text editor and Evennia must speak the same "language" so to speak. Evennia will by default first try the international `UTF-8` encoding, but you can have Evennia try any sequence of different encodings by customizing the `ENCODINGS` list in your settings file. Evennia will use the first encoding in the list that do not raise any errors. Only if none work will the server give up and return an error message.

You can often change the text editor encoding (this depends on your editor though), otherwise you need to add the editor's encoding to Evennia's `ENCODINGS` list. If you are unsure, write a test file with lots of non-ASCII letters in the editor of your choice, then import to make sure it works as it should.

More help with encodings can be found in the entry Text Encodings and also in the Wikipedia article here.

A footnote for the batch-code processor: Just because *Evennia* can parse your file and your fancy special characters, doesn't mean that *Python* allows their use. Python syntax only allows international characters inside *strings*. In all other source code only `ASCII` set characters are allowed.

Batch Command Processor

For an introduction and motivation to using batch processors, see here. This page describes the Batch-*command* processor. The Batch-*code* one is covered here.

Basic Usage

The batch-command processor is a superuser-only function, invoked by

```
> @batchcommand path.to.batchcmdfile
```

Where `path.to.batchcmdfile` is the path to a *batch-command file* with the "`.ev`" file ending. This path is given like a python path relative to a folder you define to hold your batch files, set with `BATCH_IMPORT_PATH` in your settings. Default folder is (assuming your game is in the `mygame` folder) `mygame/world`. So if you want to run the example batch file in `mygame/world/batch_cmds.ev`, you could use

```
> @batchcommand batch_cmds
```

A batch-command file contains a list of Evennia in-game commands separated by comments. The processor will run the batch file from beginning to end. Note that *it will not stop if commands in it fail* (there is no universal way for the processor to know what a failure looks like for all different commands). So keep a close watch on the output, or use *Interactive mode* (see below) to run the file in a more controlled, gradual manner.

The batch file

The batch file is a simple plain-text file containing Evennia commands. Just like you would write them in-game, except you have more freedom with line breaks.

Here are the rules of syntax of an `*.ev` file. You'll find it's really, really simple:

- All lines having the `#` (hash)-symbol *as the first one on the line* are considered *comments*. All non-comment lines are treated as a command and/or their arguments.
- Comment lines have an actual function – they mark the *end of the previous command definition*. So never put two commands directly after one another in the file - separate them with a comment, or the second of the two will be considered an argument to the first one. Besides, using plenty of comments is good practice anyway.
- A line that starts with the word `#INSERT` is a comment line but also signifies a special instruction. The syntax is `#INSERT <path.batchfile>` and tries to import a given batch-cmd file into this one. The inserted batch file (file ending `.ev`) will run normally from the point of the `#INSERT` instruction.
- Extra whitespace in a command definition is *ignored*. - A completely empty line translates in to a line break in texts. Two empty lines thus means a new paragraph (this is obviously only relevant for commands accepting such formatting, such as the `@desc` command).
- The very last command in the file is not required to end with a comment.
- You *cannot* nest another `@batchcommand` statement into your batch file. If you want to link many batch-files together, use the `#INSERT` batch instruction instead. You also cannot launch the `@batchcode` command from your batch file, the two batch processors are not compatible.

Below is a version of the example file found in `evennia/contrib/tutorial_examples/batch_cmds.ev`.

```
#
# This is an example batch build file for Evennia.
#

# This creates a red button
@create button:tutorial_examples.red_button.RedButton
# (This comment ends input for @create)
# Next command. Let's create something.
@set button/desc =
  This is a large red button. Now and then
  it flashes in an evil, yet strangely tantalizing way.

  A big sign sits next to it. It says:


-----------

```

```
 Press me!

-----------


  ... It really begs to be pressed! You
know you want to!

# This inserts the commands from another batch-cmd file named
# batch_insert_file.ev.
#INSERT examples.batch_insert_file


# (This ends the @set command). Note that single line breaks
# and extra whitespace in the argument are ignored. Empty lines
# translate into line breaks in the output.
# Now let's place the button where it belongs (let's say limbo #2 is
# the evil lair in our example)
@teleport #2
# (This comments ends the @teleport command.)
# Now we drop it so others can see it.
# The very last command in the file needs not be ended with #.
drop button
```

To test this, run `@batchcommand` on the file:

```
> @batchcommand contrib.tutorial_examples.batch_cmds
```

A button will be created, described and dropped in Limbo. All commands will be executed by the user calling the command.

> Note that if you interact with the button, you might find that its description changes, loosing your custom-set description above. This is just the way this particular object works.

Interactive mode

Interactive mode allows you to more step-wise control over how the batch file is executed. This is useful for debugging and also if you have a large batch file and is only updating a small part of it – running the entire file again would be a waste of time (and in the case of `@create`-ing objects you would to end up with multiple copies of same-named objects, for example). Use `@batchcommand` with the `/interactive` flag to enter interactive mode.

```
> @batchcommand/interactive tutorial_examples.batch_cmds
```

You will see this:

```
01/04: @create button:tutorial_examples.red_button.RedButton  (hh for help)
```

This shows that you are on the `@create` command, the first out of only four commands in this batch file. Observe that the command `@create` has *not* been actually processed at this point!

To take a look at the full command you are about to run, use `ll` (a batch-processor version of `look`). Use `pp` to actually process the current command (this will actually `@create` the button) – and make sure it worked as planned. Use `nn` (next) to go to the next command. Use `hh` for a list of commands.

If there are errors, fix them in the batch file, then use `rr` to reload the file. You will still be at the same command and can rerun it easily with `pp` as needed. This makes for a simple debug cycle. It also allows you to rerun individual troublesome commands - as mentioned, in a large batch file this can be very useful. Do note that in many cases,

commands depend on the previous ones (e.g. if `@create` in the example above had failed, the following commands would have had nothing to operate on).

Use `nn` and `bb` (next and back) to step through the file; e.g. `nn 12` will jump 12 steps forward (without processing any command in between). All normal commands of Evennia should work too while working in interactive mode.

Limitations and Caveats

The batch-command processor is great for automating smaller builds or for testing new commands and objects repeatedly without having to write so much. There are several caveats you have to be aware of when using the batch-command processor for building larger, complex worlds though.

The main issue is that when you run a batch-command script you (*you*, as in your superuser character) are actually moving around in the game creating and building rooms in sequence, just as if you had been entering those commands manually, one by one. You have to take this into account when creating the file, so that you can 'walk' (or teleport) to the right places in order.

This also means there are several pitfalls when designing and adding certain types of objects. Here are some examples:

- *Rooms that change your 'Command Set'_*: Imagine that you build a 'dark' room, which severely limits the cmdsets of those entering it (maybe you have to find the light switch to proceed). In your batch script you would create this room, then teleport to it - and promptly be shifted into the dark state where none of your normal build commands work ...
- *Auto-teleportation*: Rooms that automatically teleport those that enter them to another place (like a trap room, for example). You would be teleported away too.
- *Mobiles*: If you add aggressive mobs, they might attack you, drawing you into combat. If they have AI they might even follow you around when building - or they might move away from you before you've had time to finish describing and equipping them!

The solution to all these is to plan ahead. Make sure that superusers are never affected by whatever effects are in play. Add an on/off switch to objects and make sure it's always set to *off* upon creation. It's all doable, one just needs to keep it in mind.

Assorted notes

The fact that you build as 'yourself' can also be considered an advantage however, should you ever decide to change the default command to allow others than superusers to call the processor. Since normal access-checks are still performed, a malevolent builder with access to the processor should not be able to do all that much damage (this is the main drawback of the Batch Code Processor)

GNU Emacs users might find it interesting to use emacs' *evennia mode*. This is an Emacs major mode found in `evennia/utils/evennia-mode.el`. It offers correct syntax highlighting and indentation with `<tab>` when editing `.ev` files in Emacs. See the header of that file for installation instructions.

Batch Code Processor

For an introduction and motivation to using batch processors, see here. This page describes the Batch-*code* processor. The Batch-*command* one is covered here.

Basic Usage

The batch-command processor is a superuser-only function, invoked by

```
> @batchcode path.to.batchcodefile
```

Where `path.to.batchcodefile` is the path to a *batch-code file*. Such a file should have a name ending in "`.py`" (but you shouldn't include that in the path). The path is given like a python path relative to a folder you define to hold your batch files, set by `BATCH_IMPORT_PATH` in your settings. Default folder is (assuming your game is called "mygame") `mygame/world/`. So if you want to run the example batch file in `mygame/world/batch_code.py`, you could simply use

```
> @batchcode batch_code
```

This will try to run through the entire batch file in one go. For more gradual, *interactive* control you can use the `/interactive` switch. The switch `/debug` will put the processor in *debug* mode. Read below for more info.

The batch file

A batch-code file is a normal Python file. The difference is that since the batch processor loads and executes the file rather than importing it, you can reliably update the file, then call it again, over and over and see your changes without needing to `@reload` the server. This makes for easy testing. In the batch-code file you have also access to the following global variables:

- `caller` - This is a reference to the object running the batchprocessor.
- `DEBUG` - This is a boolean that lets you determine if this file is currently being run in debug-mode or not. See below how this can be useful.

Running a plain Python file through the processor will just execute the file from beginning to end. If you want to get more control over the execution you can use the processor's *interactive* mode. This runs certain code blocks on their own, rerunning only that part until you are happy with it. In order to do this you need to add special markers to your file to divide it up into smaller chunks. These take the form of comments, so the file remains valid Python.

Here are the rules of syntax of the batch-command `*.py` file.

- `#CODE` as the first on a line marks the start of a *code* block. It will last until the beginning of another marker or the end of the file. Code blocks contain functional python code. Each `#CODE` block will be run in complete isolation from other parts of the file, so make sure it's self-contained.
- `#HEADER` as the first on a line marks the start of a *header* block. It lasts until the next marker or the end of the file. This is intended to hold imports and variables you will need for all other blocks .All python code defined in a header block will always be inserted at the top of every `#CODE` blocks in the file. You may have more than one `#HEADER` block, but that is equivalent to having one big one. Note that you can't exchange data between code blocks, so editing a header-variable in one code block won't affect that variable in any other code block!
- `#INSERT path.to.file` will insert another batchcode (Python) file at that position.
- A `#` that is not starting a `#HEADER`, `#CODE` or `#INSERT` instruction is considered a comment.
- Inside a block, normal Python syntax rules apply. For the sake of indentation, each block acts as a separate python module.

Below is a version of the example file found in `evennia/contrib/tutorial_examples/`.

```
#
# This is an example batch-code build file for Evennia.
#
```

```
#HEADER

# This will be included in all other #CODE blocks

from evennia.utils import create, search
from evennia.contrib.tutorial_examples import red_button
from typeclasses.objects import Object

limbo = search.objects(caller, 'Limbo', global_search=True)[0]


#CODE

red_button = create.create_object(red_button.RedButton, key="Red button",
                                  location=limbo, aliases=["button"])

# caller points to the one running the script
caller.msg("A red button was created.")

# importing more code from another batch-code file
#INSERT batch_code_insert

#CODE

table = create.create_object(Object, key="Blue Table", location=limbo)
chair = create.create_object(Object, key="Blue Chair", location=limbo)

string = "A %s and %s were created."
if DEBUG:
    table.delete()
    chair.delete()
    string += " Since debug was active, " \
         "they were deleted again."
caller.msg(string % (table, chair))
```

This uses Evennia's Python API to create three objects in sequence.

Debug mode

Try to run the example script with

```
> @batchcode/debug tutorial_examples.example_batch_code
```

The batch script will run to the end and tell you it completed. You will also get messages that the button and the two pieces of furniture were created. Look around and you should see the button there. But you won't see any chair nor a table! This is because we ran this with the `/debug` switch, which is directly visible as `DEBUG==True` inside the script. In the above example we handled this state by deleting the chair and table again.

The debug mode is intended to be used when you test out a batchscript. Maybe you are looking for bugs in your code or try to see if things behave as they should. Running the script over and over would then create an ever-growing stack of chairs and tables, all with the same name. You would have to go back and painstakingly delete them later.

Interactive mode

Interactive mode works very similar to the batch-command processor counterpart. It allows you more step-wise control over how the batch file is executed. This is useful for debugging or for picking and choosing only particular blocks to run. Use `@batchcommand` with the `/interactive` flag to enter interactive mode.

```
> @batchcode/interactive tutorial_examples.batch_code
```

You should see the following:

```
01/02: red_button = create_object(red_button.RedButton, [...]          (hh for help)
```

This shows that you are on the first `#CODE` block, the first of only two commands in this batch file. Observe that the block has *not* actually been executed at this point!

To take a look at the full code snippet you are about to run, use `ll` (a batch-processor version of `look`).

```
from evennia.utils import create, search
from evennia.contrib.tutorial_examples import red_button
from typeclasses.objects Object

limbo = search.objects(caller, 'Limbo', global_search=True)[0]

red_button = create.create_object(red_button.RedButton, key="Red button",
                                  location=limbo, aliases=["button"])

# caller points to the one running the script
caller.msg("A red button was created.")
```

Compare with the example code given earlier. Notice how the content of `#HEADER` has been pasted at the top of the `#CODE` block. Use `pp` to actually execute this block (this will create the button and give you a message). Use `nn` (next) to go to the next command. Use `hh` for a list of commands.

If there are tracebacks, fix them in the batch file, then use `rr` to reload the file. You will still be at the same code block and can rerun it easily with `pp` as needed. This makes for a simple debug cycle. It also allows you to rerun individual troublesome blocks - as mentioned, in a large batch file this can be very useful (don't forget the `/debug` mode either).

Use `nn` and `bb` (next and back) to step through the file; e.g. `nn 12` will jump 12 steps forward (without processing any blocks in between). All normal commands of Evennia should work too while working in interactive mode.

Limitations and Caveats

The batch-code processor is by far the most flexible way to build a world in Evennia. There are however some caveats you need to keep in mind.

- **Safety**. Or rather the lack of it. There is a reason only *superusers* are allowed to run the batch-code processor by default. The code-processor runs **without any Evennia security checks** and allows full access to Python. If an untrusted party could run the code-processor they could execute arbitrary python code on your machine, which is potentially a very dangerous thing. If you want to allow other users to access the batch-code processor you should make sure to run Evennia as a separate and very limited-access user on your machine (i.e. in a 'jail'). By comparison, the batch-command processor is much safer since the user running it is still 'inside' the game and can't really do anything outside what the game commands allow them to.
- **You cannot communicate between code blocks**. Global variables won't work in code batch files, each block is executed as stand-alone environments. Similarly you cannot in one `#CODE` block assign to variables from the `#HEADER` block and expect to be able to read the changes from another `#CODE` block (whereas a python execution limitation, allowing this would also lead to very hard-to-debug code when using the interactive mode).

The main issue with this is when building e.g. a room in one code block and later want to connect that room with a room you built in another block. To do this, you must perform a database search for the name of the room you created (since you cannot know in advance which dbref it got assigned). This sounds iffy, but there is an easy way to handler this - use object aliases. You can assign any number of aliases to any object. Make sure that one of those aliases is unique (like "room56") and you will henceforth be able to always find it later by searching for it from other code blocks regardless of if the main name is shared with hundreds of other rooms in your world (coincidentally, this is also one way of implementing "zones", should you want to group rooms together).

- **Defining TypeClasses or treating a batchcode file like a normal Python file will lead to ruin**. Python batchcode files are syntactically valid Python modules. However, doing anything that would import them is the equivalent of running them in their entirety. Therefore you don't want to define typeclasses in them, because any time Evennia would import the module to find the class, a whole host of new objects would be created by your batch code.
- **Code that relies on the batch file's real file path will fail**. Batch code files are chopped up into code snippets and are executed based on the resulting strings and a custom dictionary context. This means that the code will lose association with any file it was once a part of.

Building Permissions

OBS: This gives only a brief introduction to the access system. Locks and permissions are fully detailed here.

The super user

There are strictly speaking two types of users in Evennia, the *super user* and everyone else. The superuser is the first user you create, object `#1`. This is the all-powerful server-owner account. Technically the superuser not only has access to everything, it *bypasses* the permission checks entirely. This makes the superuser impossible to lock out, but makes it unsuitable to actually play-test the game's locks and restrictions with (see `@quell` below). Usually there is no need to have but one superuser.

Assigning permissions

Whereas permissions can be used for anything, those put in `settings.PERMISSION_HIERARCHY` will have a ranking relative each other as well. We refer to these types of permissions as *hierarchical permissions*. When building locks to check these permissions, the `perm()` lock function is used. By default Evennia creates the following hierarchy (spelled exactly like this):

1. **Immortals** basically have the same access as superusers except that they do *not* sidestep the Permission system. Assign only to really trusted server-admin staff since this level gives access both to server reload/shutdown functionality as well as (and this may be more critical) gives access to the all-powerful `@py` command that allows the execution of arbitrary Python code on the command line.
2. **Wizards** can do everything *except* affecting the server functions themselves. So a wizard couldn't reload or shutdown the server for example. They also cannot execute arbitrary Python code on the console or import files from the hard drive.
3. **Builders** - have all the build commands, but cannot affect other players or mess with the server.
4. **PlayerHelpers** are almost like a normal *Player*, but they can also add help files to the database.
5. **Players** is the default group that new players end up in. A new player have permission to use tells and to use and create new channels.

A user having a certain level of permission automatically have access to locks specifying access of a lower level.

To assign a new permission from inside the game, you need to be able to use the `@perm` command. This is an *Immortal*-level command, but it could in principle be made lower-access since it only allows assignments equal or lower to your current level (so you cannot use it to escalate your own permission level). So, assuming you yourself have *Immortal* access (or is superuser), you assign a new player "Tommy" to your core staff with the command

```
@perm/player Tommy = Immortals
```

or

```
@perm *Tommy = Immortals
```

We use a switch or the `*name` format to make sure to put the permission on the *Player* and not on any eventual *Character* that may also be named "Tommy". This is usually what you want since the Player will then remain an Immortal regardless of which Character they are currently controlling. To limit permission to a per-Character level you should instead use *quelling* (see below). Also remember that the access group you add is by default in plural ("Immortals", not "Immortal") - since permissions can have any name, the system will not complain if you assign the wrong one.

Quelling your permissions

When developing it can be useful to check just how things would look had your permission-level been lower. For this you can use *quelling*. Normally, when you puppet a Character you are using your Player-level permission. So even if your Character only has *Players* level permissions, your *Immortals*-level Player will take precedence. With the `@quell` command you can change so that the Character's permission takes precedence instead:

```
@quell
```

This will allow you to test out the game using the current Character's permission level. A developer or builder can thus in principle maintain several test characters, all using different permission levels. Note that you cannot escalate your permissions this way; If the Character happens to have a *higher* permission level than the Player, the *Player's* (lower) permission will still be used.

CHAPTER 10

Developer Basics

This chapter gives an introduction to coding with Evennia.

Coding Introduction

Evennia allows for a lot of freedom when designing your game - but to code efficiently you still need to adopt some best practices as well as find a good place to start to learn.

Here are some pointers to get you going.

Explore Evennia interactively

When new to Evennia it can be hard to find things or figure out what is available. Evennia offers a special interactive python shell that allows you to experiment and try out things. It's recommended to use ipython for this since the vanilla python prompt is very limited. Here are some simple commands to get started:

```
# [open a new console/terminal]
# [activate your evennia virtualenv in this console/terminal]
pip install ipython    # [only needed the first time]
cd mygame
evennia shell
```

This will open an Evennia-aware python shell (using ipython). From within this shell, try

```
import evennia
evennia.<TAB>
```

That is, enter `evennia.` and press the `<TAB>` key. This will show you all the resources made available at the top level of Evennia's "flat API". See the flat API page for more info on how to explore it efficiently.

You can complement your exploration by peeking at the sections of the much more detailed Developer Central. The Tutorials section also contains a growing collection of system- or implementation-specific help.

Use a python syntax checker

Evennia works by importing your own modules and running them as part of the server. Whereas Evennia should just gracefully tell you what errors it finds, it can nevertheless be a good idea for you to check your code for simple syntax errors *before* you load it into the running server. There are many python syntax checkers out there. A fast and easy one is pyflakes, a more verbose one is pylint. You can also check so that your code looks up to snuff using pep8. Even with a syntax checker you will not be able to catch every possible problem - some bugs or problems will only appear when you actually run the code. But using such a checker can be a good start to weed out the simple problems.

Plan before you code

Before you start coding away at your dream game, take a look at our Game Planning page. It might hopefully help you avoid some common pitfalls and time sinks.

Code in your game folder, not in the evennia/ repository

As part of the Evennia setup you will create a game folder to host your game code. This is your home. You should *never* need to modify anything in the `evennia` library (anything you download from us, really). You import useful functionality from here and if you see code you like, copy&paste it out into your game folder and edit it there.

If you find that Evennia doesn't support some functionality you need, make a Feature Request about it. Same goes for bugs. If you add features or fix bugs yourself, please consider Contributing your changes upstream!

Learn to read tracebacks

Python is very good at reporting when and where things go wrong. A *traceback* shows everything you need to know about crashing code. The text can be pretty long, but you usually are only interested in the last bit, where it says what the error is and at which module and line number it happened - armed with this info you can resolve most problems.

Evennia will usually not show the full traceback in-game though. Instead the server outputs errors to the terminal/console from which you started Evennia in the first place. If you want more to show in-game you can add `IN_GAME_ERRORS = True` to your settings file. This will echo most (but not all) tracebacks both in-game as well as to the terminal/console. This is a potential security problem though, so don't keep this active when your game goes into production.

> A common confusing error is finding that objects in-game are suddenly of the type `DefaultObject` rather than your custom typeclass. This happens when you introduce a critical Syntax error to the module holding your custom class. Since such a module is not valid Python, Evennia can't load it at all. Instead of crashing, Evennia will then print the full traceback to the terminal/console and temporarily fall back to the safe `DefaultObject` until you fix the problem and reload.

Docs are here to help you

Some people find reading documentation extremely dull and shun it out of principle. That's your call, but reading docs really *does* help you, promise! Evennia's documentation is pretty thorough and knowing what is possible can often give you a lot of new cool game ideas. That said, if you can't find the answer in the docs, don't be shy to ask questions! The discussion group and the irc chat are also there for you.

The most important point

And finally, of course, have fun!

Version Control

<<<<<<< HEAD
“_
Version control software allows you to track the changes you make to your code, as well as being able to easily backtrack these changes, share your development efforts and more. Even if you are not contributing to Evennia itself, and only wish to develop your own MU* using Evennia, having a version control system in place is a good idea (and standard coding practice). For an introduction to the concept start with the Wikipedia article here. Note that this page deals with commands for Linux operating systems, and the steps below may vary for other systems, however where possible links will be provided for alternative instructions.

For more help on using Git, please refer to the Official Github documentation.

Setting up Git

If you have gotten Evennia installed, you will have Git already and can skip to **Step 2** below. Otherwise you will need to install Git on your platform. You can find expanded instructions for installation here.

Step 1: Install Git

- **Fedora Linux**

```
yum install git-core
```

- **Debian Linux** *(Ubuntu, Linux Mint, etc.)*

```
apt-get install git
```

- **Windows**: It is recommended to use Git for Windows.
- **Mac**: Mac platforms offer two methods for installation, one via MacPorts, which you can find out about here, or you can use the Git OSX Installer.

Step 2: Define user/e-mail Settings for Git

To avoid a common issue later, you will need to set a couple of settings; first you will need to tell Git your username, followed by your e-mail address, so that when you commit code later you will be properly credited.

1. Set the default name for git to use when you commit:

```
git config --global user.name "Your Name Here"
```

2. Set the default email for git to use when you commit:

```
git config --global user.email "your_email@example.com"
```

Forking from Evennia

Step 1: Fork the evennia/master repository

> Before proceeding with the following step, make sure you have registered and created an account on Github.com. This is necessary in order to create a fork of Evennia's master repository, and to push your commits to your fork either for yourself or for contributing to Evennia.

A *fork* is a clone of the master repository that you can make your own commits and changes to. At the top of this page, click the "Fork" button, as it appears below.

Step 2: Clone your fork

The fork only exists online as of yet. In a terminal, change your directory to the folder you wish to develop in. From this directory run the following command:

```
git clone https://github.com/yourusername/evennia.git
```

This will download your fork to your computer. It creates a new folder `evennia/` at your current location.

Step 3: Configure remotes

A *remote* is a repository stored on another computer, in this case on GitHub's server. When a repository is cloned, it has a default remote called `origin`. This points to your fork on GitHub, not the original repository it was forked from. To easily keep track of the original repository (that is, Evennia's official repository), you need to add another remote. The standard name for this remote is "upstream".

Below we change the active directory to the newly cloned "evennia" directory and then assign the original Evennia repository to a remote called "upstream":

```
cd evennia
git remote add upstream https://github.com/evennia/evennia.git
```

Working with your fork

Making a work branch

> A *branch* is a separate instance of your code. Changes you do to code in a branch does not affect that in other branches (so if you for example add/commit a file to one branch and then switches to another branch, that file will be gone until you switch back to the first branch again). One can switch between branches at will and create as many branches as one needs for a given project. The content of branches can also be merged together or deleted without affecting other branches. This is not only a common way to organize development but also to test features without messing with existing code.

The default *branch* of git is called the "master" branch. We will let our master branch be our "clean" Evennia install - this is a good debugging practice since it allows us to know if a bug you find is due to your changes or also visible in the core server. We'll develop our game in another branch instead, let's call it "mygame".

```
git checkout -b mygame
```

This command will checkout and automatically create the new branch `mygame` on your machine. You can see which branch you are on with `git branch` and change between different branches with `git checkout <branchname>`.

Making a branch for Evennia-fixes

If you want to contribute fixes to Evennia itself, it's a good idea to keep those separate from your own game implementation. This makes it easier for Evennia developers to later pull in only the changes that are relevant. Get back to the master branch (`git checkout master`) and create a new "myfixes" branch:

```
git checkout -b myfixes
```

Tracking files

When working on your code or fix bugs in your local branches you may end up creating new files. If you do you must tell Git to track them by using the add command:

```
git add <filename>
```

You can check the current status of version control with `git status`. This will show if you have any modified, added or otherwise changed files. Some files, like database files, logs and temporary PID files are usually *not* tracked in version control. These should have a question mark in front of them.

Controlling tracking

In order to not track system settings so that you and/or your team can pull without worries of directory structure it is recommended to append your `.gitignore` file located in the root directory `evennia`.

Be sure that you are NOT in the branch `master` when appending this file or updates will not download due to the folder `src` being ignored. Open `.gitignore`, which is a hidden file (note the period at the beginning of the file name). At the end of the file add the following:

```
# Custom
settings.py
*.log
*.log.old
src
```

Now you can pull and install without having to edit the files for system-specific data. If `settings.py` does need to be modified, simply share with your team what needs to be edited. The reason for ignoring it is because different servers may need different settings, especially if you are working in a team and have a remote server to worry about.

Committing your Code

> Committing means storing the current snapshot of your code within git. This creates a "save point" or "history" of your development process. You can later jump back and forth in your history, for example to figure out just when a bug was introduced or see what results the code used to produce compared to now.

It's usually a good idea to commit your changes often. Committing is fast and local only - you will never commit anything online at this point. To commit your changes, use

```
git commit --all
```

This will save all changes you made since last commit. The command will open a text editor where you can add a message detailing the changes you've made. Make it brief but informative. You can see the history of commits with `git log`. If you don't want to use the editor you can set the message directly by using the `-m` flag:

```
git commit --all -m "This fixes a bug in the combat code."
```

Changing your mind

If you have non-committed changes that you realize you want to throw away, you can do the following:

```
git checkout <file to revert>
```

This will revert the file to the state it was in at your last `commit`, throwing away the changes you did to it since. It's a good way to make wild experiments without having to remember just what you changed. If you do `git checkout .` you will throw away *all* changes since the last commit.

Updating with upstream changes

When Evennia's official repository updates, first make sure to commit all your changes to your branch and then checkout the "clean" master branch:

```
git commit --all
git checkout master
```

Pull the latest changes from upstream:

```
git pull upstream master
```

This should sync your local master branch with upstream Evennia's master branch (where our development happens). Now we go back to our own work-branch and *merge* the updated master into our branch.

```
git checkout mygame
git merge master
```

If everything went well, your `mygame` branch will now have the latest version of Evennia. To update also your `myfixes` branch just do

```
git checkout myfixes
git merge master
```

Use `git log` to see what has changed. You may need to restart the server or run `manage.py migrate` if the database schema changed (this will be seen in the commit log and on the mailing list). See the Git manuals for learning more about useful day-to-day commands, and special situations such as dealing with merge collisions.

Sharing your Code Publicly

Up to this point your `mygame` and `myfixes` branches only exist on your local computer. No one else can see them. If you want a copy of those branches to also appear in your online fork on github, make sure to have checked out your "mygame" branch and then run the following:

```
git push -u mygame
```

This will create a new *remote branch* named "mygame" in your online repository. Henceforth you can just use `git push` from your mygame branch to push your changes online. This is a great way to keep your source backed-up and accessible. Remember though that unless you have paid for a "private" repository at Github everyone will be able to browse and download your code (same way as you can with Evennia itself).

Committing fixes to Evennia

Let's say you found a bug in Evennia and want to contribute with a fix. We will assume you have set up your local repository as outlined in the previous sections. We will assume you do your fixing in the "myfixes" branch (but you might as well consider having a new branch named appropriately for every feature you want to contribute).

First commit any changes you may have made elsewhere and then update the master branch and your "myfixes" branch to the latest evennia version:

```
git checkout master
git pull upstream
git checkout myfixes
git merge master
```

Now you fix and test things in your "myfixes" branch, committing as you go:

```
git commit --all -m "This fixes issue #124."
```

Make sure to always make clear and descriptive commit messages so it's easy to see what you intended. If you implement multiple separate features/bug-fixes, split them into separate commits. You can do any number of commits as you work. Once you are at a stage where you want to show the world what you did, push all the so-far committed changes to your online clone, in a new remote branch:

```
git push -u myfixes
```

This only needs to be done once. If you already created the remote branch earlier, just stand in your "myfixes" branch and do `git push`.

You next need to tell the Evennia developers that they should merge your brilliant changes into Evennia proper. Create a pull request and follow the instructions. Make sure to specifically select your myfixes branch to be the source of the merge. Evennia developers will then be able to examine your request and merge it if it is deemed suitable.

Sharing your Code Privately

Creating a publicly visible online clone might not be what you want for all parts of your development process - you may prefer a more private venue when sharing your revolutionary work with your team.

GitHub offers private repositories at a cost. Alternatively you could host your code on BitBucket, which offers free private repositories as long as your development team is not too big.
‘ <A-tutorial-on-using-version-control-for-your-own-sanity-and-safety.>‘__

Version control software allows you to track the changes you make to your code, as well as being able to easily backtrack these changes, share your development efforts and more. Even if you are not contributing to Evennia itself, and only wish to develop your own MU* using Evennia, having a version control system in place is a good idea (and standard coding practice). For an introduction to the concept, start with the Wikipedia article here. Evennia uses the version control system Git and this is what will be covered henceforth. Note that this page also deals with commands for Linux operating systems, and the steps below may vary for other systems, however where possible links will be provided for alternative instructions.

For more help on using Git, please refer to the Official GitHub documentation.

Setting up Git

If you have gotten Evennia installed, you will have Git already and can skip to **Step 2** below. Otherwise you will need to install Git on your platform. You can find expanded instructions for installation here.

Step 1: Install Git

- **Fedora Linux**

```
yum install git-core
```

- **Debian Linux** *(Ubuntu, Linux Mint, etc.)*

```
apt-get install git
```

- **Windows**: It is recommended to use Git for Windows.
- **Mac**: Mac platforms offer two methods for installation, one via MacPorts, which you can find out about here, or you can use the Git OSX Installer.

Step 2: Define user/e-mail Settings for Git

To avoid a common issue later, you will need to set a couple of settings; first you will need to tell Git your username, followed by your e-mail address, so that when you commit code later you will be properly credited.

> Note that your commit information will be visible to everyone if you ever contribute to Evennia or use an online service like github to host your code. So if you are not comfortable with using your real, full name online, put a nickname here.

1. Set the default name for git to use when you commit:

```
git config --global user.name "Your Name Here"
```

2. Set the default email for git to use when you commit:

```
git config --global user.email "your_email@example.com"
```

Putting your game folder under version control

> Note: The game folder's version control is completely separate from Evennia's repository.

After you have set up your game you will have created a new folder to host your particular game (let's call this folder `mygame` for now).

This folder is *not* under version control at this point.

```
git init mygame
```

Your mygame folder is now ready for version control! Now add all the content and make a first commit:

```
cd mygame
git add *
git commit -m "Initial commit"
```

Read on for help on what these commands do.

Tracking files

When working on your code or fix bugs in your local branches you may end up creating new files. If you do you must tell Git to track them by using the add command:

```
git add <filename>
```

You can check the current status of version control with `git status`. This will show if you have any modified, added or otherwise changed files. Some files, like database files, logs and temporary PID files are usually *not* tracked in version control. These should have a question mark in front of them.

Controlling tracking

You will notice that some files are not covered by your git version control, notably your settings file (`mygame/server/conf/settings.py`) and your sqlite3 database file `mygame/server/evennia.db3`. This is controlled by the hidden file `mygame/.gitignore`. Evennia creates this file as part of the creation of your game directory. Everything matched in this file will be ignored by GIT. If you want to, for example, include your settings file for collaborators to access, remove that entry in `.gitignore`.

Committing your Code

> Committing means storing the current snapshot of your code within git. This creates a "save point" or "history" of your development process. You can later jump back and forth in your history, for example to figure out just when a bug was introduced or see what results the code used to produce compared to now.

It's usually a good idea to commit your changes often. Committing is fast and local only - you will never commit anything online at this point. To commit your changes, use

```
git commit --all
```

This will save all changes you made since last commit. The command will open a text editor where you can add a message detailing the changes you've made. Make it brief but informative. You can see the history of commits with `git log`. If you don't want to use the editor you can set the message directly by using the `-m` flag:

```
git commit --all -m "This fixes a bug in the combat code."
```

Changing your mind

If you have non-committed changes that you realize you want to throw away, you can do the following:

```
git checkout <file to revert>
```

This will revert the file to the state it was in at your last `commit`, throwing away the changes you did to it since. It's a good way to make wild experiments without having to remember just what you changed. If you do `git checkout .` you will throw away *all* changes since the last commit.

Pushing your code online

So far your code is only located on your private machine. A good idea is to back it up online. The easiest way to do this is to push it to your own remote repository on GitHub.

1. Make sure you have your game directory setup under git version control as described above. Make sure to commit any changes.

2. Create a new, empty repository on Github. Github explains how here (do *not* "Initialize the repository with a README" or else you'll create unrelated histories).
3. From your local game dir, do `git remote add origin <github URL>` where `<github URL>` is the URL to your online repo. This tells your game dir that it should be pushing to the remote online dir.
4. `git remote -v` to verify the online dir.
5. `git push origin master` now pushes your game dir online so you can see it on github.com.

You can commit your work locally (`git commit -all -m "Make a change that ..."`) as many times as you want. When you want to push those changes to your online repo, you do `git push`. You can also `git clone <url_to_online_repo>` from your online repo to somewhere else (like your production server) and henceforth do `git pull` to update that to the latest thing you pushed.

Note that GitHub's repos are, by default publicly visible by all. Creating a publicly visible online clone might not be what you want for all parts of your development process - you may prefer a more private venue when sharing your revolutionary work with your team.

GitHub offers private repositories at a cost. Alternatively you could host your code on BitBucket, which offers free private repositories as long as your development team is not too big.

Forking Evennia

This helps you set up an online *fork* of Evennia so you can easily commit fixes and help with upstream development.

Step 1: Fork the evennia/master repository

> Before proceeding with the following step, make sure you have registered and created an account on GitHub.com. This is necessary in order to create a fork of Evennia's master repository, and to push your commits to your fork either for yourself or for contributing to Evennia.

A *fork* is a clone of the master repository that you can make your own commits and changes to. At the top of this page, click the "Fork" button, as it appears below.

Step 2: Clone your fork

The fork only exists online as of yet. In a terminal, change your directory to the folder you wish to develop in. From this directory run the following command:

```
git clone https://github.com/yourusername/evennia.git
```

This will download your fork to your computer. It creates a new folder `evennia/` at your current location.

Step 3: Configure remotes

A *remote* is a repository stored on another computer, in this case on GitHub's server. When a repository is cloned, it has a default remote called `origin`. This points to your fork on GitHub, not the original repository it was forked from. To easily keep track of the original repository (that is, Evennia's official repository), you need to add another remote. The standard name for this remote is "upstream".

Below we change the active directory to the newly cloned "evennia" directory and then assign the original Evennia repository to a remote called "upstream":

```
cd evennia
git remote add upstream https://github.com/evennia/evennia.git
```

Working with your fork

> A *branch* is a separate instance of your code. Changes you do to code in a branch does not affect that in other branches (so if you for example add/commit a file to one branch and then switches to another branch, that file will be gone until you switch back to the first branch again). One can switch between branches at will and create as many branches as one needs for a given project. The content of branches can also be merged together or deleted without affecting other branches. This is not only a common way to organize development but also to test features without messing with existing code.

The default *branch* of git is called the "master" branch. As a rule of thumb, you should *never* make modifications directly to your local copy of the master branch. Rather keep the master clean and only update it by pulling our latest changes to it. Any work you do should instead happen in a local, other branches.

Making a work branch

```
git checkout -b myfixes
```

This command will checkout and automatically create the new branch `myfixes` on your machine. If you stared out in the master branch, *myfixes* will be a perfect copy of the master branch. You can see which branch you are on with `git branch` and change between different branches with `git checkout <branchname>`.

Branches are fast and cheap to create and manage. It is common practice to create a new branch for every bug you want to work on or feature you want to create, then create a *pull request* for that branch to be merged upstream (see below). Not only will this organize your work, it will also make sure that *your* master branch version of Evennia is always exactly in sync with the upstream version's master branch.

Updating with upstream changes

When Evennia's official repository updates, first make sure to commit all your changes to your branch and then checkout the "clean" master branch:

```
git commit --all
git checkout master
```

Pull the latest changes from upstream:

```
git pull upstream master
```

This should sync your local master branch with upstream Evennia's master branch (where our development happens). Now we go back to our own work-branch (let's say it's still called "myfixes") and *merge* the updated master into our branch.

```
git checkout myfixes
git merge master
```

If everything went well, your `myfixes` branch will now have the latest version of Evennia merged with whatever changes you have done. Use `git log` to see what has changed. You may need to restart the server or run `manage.py migrate` if the database schema changed (this will be seen in the commit log and on the mailing list). See the Git manuals for learning more about useful day-to-day commands, and special situations such as dealing with merge collisions.

Sharing your Code Publicly

Up to this point your `myfix` branch only exists on your local computer. No one else can see it. If you want a copy of this branch to also appear in your online fork on GitHub, make sure to have checked out your "myfix" branch and then run the following:

```
git push -u origin myfix
```

This will create a new *remote branch* named "myfix" in your online repository (which is refered to as "origin" by default); the `-u` flag makes sure to set this to the default push location. Henceforth you can just use `git push` from your myfix branch to push your changes online. This is a great way to keep your source backed-up and accessible. Remember though that unless you have paid for a "private" repository at GitHub everyone will be able to browse and download your code (same way as you can with Evennia itself).

Note: If you hadn't setup a public key on GitHub or aren't asked for a username/password, you might get an error ''403: Forbidden Access'' at this stage. In that case, some users have reported that the workaround is to create a file ''.netrc'' under your home directory and add your credentials there:

```
machine github.com
login <my_github_username>
password <my_github_password>
```

Committing fixes to Evennia

Contributing can mean both bug-fixes or adding new features to Evennia. Please note that if your change is not already listed and accepted in the Issue Tracker, it is recommended that you first hit the developer mailing list or IRC chat to see beforehand if your feature is deemed suitable to include as a core feature in the engine. When it comes to bug-fixes, other developers may also have good input on how to go about resolving the issue.

To contribute you need to have *forked Evennia* first. As described above you should do your modification in a separate local branch (not in the master branch). This branch is what you then present to us (as a *Pull request*, PR, see below). We can then merge your change into the upstream master and you then do `git pull` to update master usual. Now that the master is updated with your fixes, you can safely delete your local work branch. Below we describe this work flow.

First update the Evennia master branch to the latest Evennia version:

```
git checkout master
git pull upstream
```

Next, create a new branch to hold your contribution. Let's call it the "fixing_strange_bug" branch:

```
git checkout -b fixing_strange_bug
```

It is wise to make separate branches for every fix or series of fixes you want to contribute. You are now in your new `fixing_strange_bug` branch. You can list all branches with `git branch` and jump between branches with `git checkout <branchname>`. Code and test things in here, committing as you go:

```
git commit --all -m "Fix strange bug in look command. Resolves #123."
```

You can make multiple commits if you want, depending on your work flow and progress. Make sure to always make clear and descriptive commit messages so it's easy to see what you intended. To refer to, say, issue number 123, write `#123`, it will turn to a link on GitHub. If you include the text "Resolves #123", that issue will be auto-closed on GitHub if your commit gets merged into main Evennia.

> If you refer to in-game commands that start with `@`(such as `@examine`), please put them in backticks ‘, for example ‘`@examine`‘. The reason for this is that GitHub uses `@username` to refer to GitHub users, so if you forget the ticks, any user happening to be named `examine` will get a notification ....

If you implement multiple separate features/bug-fixes, split them into different branches if they are very different and should be handled as separate PRs. You can do any number of commits to your branch as you work. Once you are at a stage where you want to show the world what you did you might want to consider making it clean for merging into Evennia's master branch by using git rebase (this is not always necessary, and if it sounds too hard, say so and we'll handle it on our end).

Once you are ready, push your work to your online Evennia fork on github, in a new remote branch:

```
git push -u origin fixing_strange_bug
```

The `-u` flag is only needed the first time - this tells GIT to create a remote branch. If you already created the remote branch earlier, just stand in your `fixing_strange_bug` branch and do `git push`.

Now you should tell the Evennia developers that they should consider merging your brilliant changes into Evennia proper. Create a pull request and follow the instructions. Make sure to specifically select your `fixing_strange_bug` branch to be the source of the merge. Evennia developers will then be able to examine your request and merge it if it's deemed suitable.

Once your changes have been merged into Evennia your local `fixing_strange_bug` can be deleted (since your changes are now available in the "clean" Evennia repository). Do

```
git branch -D fixing_strange_bug
```

to delete your work branch. Update your master branch ('checkout master`and then`git pull‘) and you should get your fix back, now as a part of official Evennia!

GIT tips and tricks

Some of the GIT commands can feel a little long and clunky if you need to do them often. Luckily you can create aliases for those. Here are some useful commands to run:

```
# git st
# - view brief status info
git config --global alias.st 'status -s'
```

```
# git cl
# - clone a repository
git config --global alias.cl clone
```

```
# git cma "commit message"
# - commit all changes without opening editor for message
git config --global alias.cma 'commit -a -m'
```

```
# git ca
# - amend text to your latest commit message
git config --global alias.ca 'commit --amend'
```

```
# git fl
# - file log; shows diffs of files in latest commits
git config --global alias.fl 'log -u'
```

```
# git co [branchname]
# - checkout
git config --global alias.co checkout
```

```
# git br <branchname>
# - create branch
git config --global alias.br branch
```

```
# git ls
# - view log tree
git config --global alias.ls 'log --pretty=format:"%C(green)%h\ %C(yellow)[%ad]%Cred
↪%d\ %Creset%s%Cblue\ [%cn]" --decorate --date=relative --graph'
```

```
# git diff
# - show current uncommitted changes
git config --global alias.diff 'diff --word-diff'
```

```
# git grep <query>
# - search (grep) codebase for a search criterion
git config --global alias.grep 'grep -Ii'
```

To get a further feel for GIT there is also a good youtube talk about it - it's a bit long but it will help you understand the underlying ideas behind GIT
(which in turn makes it a lot more intuitive to use).

Quirks

This is a list of various quirks or common stumbling blocks that people often ask about or report when using (or trying to use) Evennia. They are not bugs.

Forgetting to use @reload to see changes to your typeclasses

Firstly: Reloading the server is a safe and usually quick operation which will *not* disconnect any players.

New users tend to forget this step. When editing source code (such as when tweaking typeclasses and commands or adding new commands to command sets) you need to either use the in-game `@reload` command or, from the command line do `python evennia.py reload` before you see your changes.

Web admin to create new Player

If you use the default login system and are trying to use the Web admin to create a new Player account, you need to consider which `MULTIPLAYER_MODE` you are in. If you are in `MULTIPLAYER_MODE 0` or `1`, the login system expects each Player to also have a Character object named the same as the Player - there is no character creation screen by default. If using the normal mud login screen, a Character with the same name is automatically created and connected to your Player. From the web interface you must do this manually.

So, when creating the Player, make sure to also create the Character *from the same form* as you create the Player from. This should set everything up for you. Otherwise you need to manually set the "player" property on the Character and the "character" property on the Player to point to each other. You must also set the lockstring of the Character to allow the Player to "puppet" this particular character.

Mutable attributes and their connection to the database

When storing a mutable object (usually a list or a dictionary) in an Attribute

```
object.db.mylist = [1,2,3]
```

you should know that the connection to the database is retained also if you later extract that Attribute into another variable (what is stored and retrieved is actually a `PackedList` or a `PackedDict` that works just like their namesakes except they save themselves to the database when changed). So if you do

```
alist = object.db.mylist
alist.append(4)
```

this updates the database behind the scenes, so both `alist` and `object.db.mylist` are now `[1,2,3,4]`

If you don't want this, convert the mutable object to its normal Python form.

```
blist = list(object.db.mylist)
blist.append(4)
```

The property `blist` is now `[1,2,3,4]` whereas `object.db.mylist` remains unchanged. You'd need to explicitly re-assign it to the `mylist` Attribute in order to update the database. If you store nested mutables you only need to convert the "outermost" one in order to "break" the connection to the database like this. See Attributes for more details.

Commands are matched by name *or* alias

When merging command sets it's important to remember that command objects are identified *both* by key *or* alias. So if you have a command with a key `look` and an alias `ls`, introducing another command with a key `ls` will be assumed by the system to be *identical* to the first one. This usually means merging cmdsets will overload one of them depending on priority. Whereas this is logical once you know how command objects are handled, it may be confusing if you are just looking at the command strings thinking they are parsed as-is.

Objects turning to `DefaultObject`

A common confusing error for new developers is finding that one or more objects in-game are suddenly of the type `DefaultObject` rather than the typeclass you wanted it to be. This happens when you introduce a critical Syntax error to the module holding your custom class. Since such a module is not valid Python, Evennia can't load it at all to get to the typeclasses within. To keep on running, Evennia will solve this by printing the full traceback to the terminal/console and temporarily fall back to the safe `DefaultObject` until you fix the problem and reload. Most errors of this kind will be caught by any good text editors. Keep an eye on the terminal/console during a reload to catch such errors - you may have to scroll up if your window is small.

Known upstream bugs

There is currently (Autumn 2016) a bug in the `zope.interface` installer on some Linux Ubuntu distributions (notably Ubuntu 16.04 LTS). Zope is a dependency of Twisted. The error manifests in the server not starting with an error that `zope.interface` is not found even though `pip list` shows it's installed. The reason is a missing empty `__init__.py` file at the root of the zope package. If the virtualenv is named "pyenv" as suggested in the Getting Started instructions, use the following command to fix it:

```
touch pyenv/local/lib/python2.7/site-packages/zope/__init__.py
```

This will create the missing file and things should henceforth work correctly.

Licensing

Evennia is licensed under the very friendly BSD (3-clause) license. You can find the license as LICENSE.txt in the Evennia repository's root.

Q: When creating a game using Evennia, what does the license permit me to do with it?

A: It's your own game world to do with as you please! Keep it to yourself or re-distribute it under another license of your choice - or sell it and become filthy rich for all we care.

Q: I have modified the Evennia library itself, what does the license say about that?

A: Our license allows you to do whatever you want with your modified Evennia, including re-distributing or selling it, as long as you include our license and copyright info found in `LICENSE.txt` along with your distribution.

... Of course, if you fix bugs or add some new snazzy feature we *softly nudge* you to make those changes available so they can be added to the core Evennia package for everyone's benefit. The license doesn't *require* you to do it, but that doesn't mean we won't still greatly appreciate it if you do!

Q: Can I re-distribute the Evennia server package along with my custom game implementation?

A: Sure. As long as the text in `LICENSE.txt` is included.

Q: What about Contributions?

The contributions in `evennia/evennia/contrib` are considered to be released under the same license as Evennia itself, unless the individual contributor has specifically defined otherwise.

CHAPTER 11

Server Components

This chapter details the inner workings of Evennia. It is useful both for game developers and for developers working on Evennia itself.

Directory Overview

This is an overview of the directories relevant to Evennia coding.

The Game directory

The game directory is created with `evennia --init <name>`. In the Evennia documentation we always assume it's called `mygame`. Apart from the `server/` subfolder within, you could reorganize this folder if you preferred a different code structure for your game.

- `mygame/`
- `commands/` - Overload default Commands or add your own Commands/Command sets here.
- `server/` - The structure of this folder should not change since Evennia expects it.
  - `` `conf/``_ - All server configuration files sits here. The most important file is ``server.conf. ``
  - `logs/` - Portal log files are stored here (Server is logging to the terminal by default)
- `typeclasses/` - this folder contains empty templates for overloading default game entities of Evennia. Evennia will automatically use the changes in those templates for the game entities it creates.
- `web/` - This holds the Web features of your game.
- `world/` - this is a "miscellaneous" folder holding everything related to the world you are building, such as build scripts and rules modules that don't fit with one of the other folders.

Evennia library layout:

If you cloned the GIT repo following the instructions, you will have a folder named `evennia`. The top level of it contains Python package specific stuff such as a readme file, `setup.py` etc. It also has two subfolders`bin/` and `evennia/` (again).

The `bin/` directory holds OS-specific binaries that will be used when installing Evennia with `pip` as per the Getting started instructions. The library itself is in the `evennia` subfolder. From your code you will access this subfolder simply by `import evennia`.

- evennia
- **‘‘‘__init__.py‘‘‘_** - The “flat API” of Evennia resides here.
- **‘‘‘commands/‘‘‘_** - The command parser and handler.
  - `default/` - The default commands and cmdsets.
- **‘‘‘comms/‘‘‘_** - Systems for communicating in-game.
- `contrib/` - Optional plugins too game-specific for core Evennia.
- `game_template/` - Copied to become the “game directory” when using `evennia --init`.
- **‘‘‘help/‘‘‘_** - Handles the storage and creation of help entries.
- `locale/` - Language files (i18n).
- **‘‘‘locks/‘‘‘_** - Lock system for restricting access to in-game entities.
- **‘‘‘objects/‘‘‘_** - In-game entities (all types of items and Characters).
- **‘‘‘players/‘‘‘_** - Out-of-game Session-controlled entities (players, bots etc)
- **‘‘‘scripts/‘‘‘_** - Out-of-game entities equivalence to Objects, also with timer support.
- **‘‘‘server/‘‘‘_** - Core server code and Session handling.
  - `portal/` - Portal proxy and connection protocols.
- `` `settings_default.py```_ - Root settings of Evennia. Copy settings from here to ``mygame/server/settings.py `` file.
- **‘‘‘typeclasses/‘‘‘_** - Abstract classes for the typeclass storage and database system.
- **‘‘‘utils/‘‘‘_** - Various miscellaneous useful coding resources.
- **‘‘‘web/‘‘‘_** - Web resources and webserver. Partly copied into game directory on initialization.

All directories contain files ending in `.py`. These are Python *modules* and are the basic units of Python code. The roots of directories also have (usually empty) files named `__init__.py`. These are required by Python so as to be able to find and import modules in other directories. When you have run Evennia at least once you will find that there will also be `.pyc` files appearing, these are pre-compiled binary versions of the `.py` files to speed up execution.

The root of the `evennia` folder has an `__init__.py` file containing the “flat API”. This holds shortcuts to various subfolders in the evennia library. It is provided to make it easier to find things; it allows you to just import `evennia` and access things from that rather than having to import from their actual locations inside the source tree.

Server Conf

Evennia runs out of the box without any changes to its settings. But there are several important ways to customize the server and expand it with your own plugins.

Settings file

The "Settings" file referenced throughout the documentation is the file `mygame/server/conf/settings.py`. This is automatically created on the first run of `evennia --init` (see the Getting Started page).

Your new `settings.py` is relatively bare out of the box. Evennia's core settings file is actually evennia/settings_default.py and is considerably more extensive (it is also heavily documented so you should refer to this file directly for the available settings).

Since `mygame/server/conf/settings.py` is a normal Python module, it simply imports `evennia/settings_default.py` into itself at the top.

This means that if any setting you want to change were to depend on some *other* default setting, you might need to copy & paste both in order to change them and get the effect you want (for most commonly changed settings, this is not something you need to worry about).

You should never edit `evennia/settings_default.py`. Rather you should copy&paste the select variables you want to change into your `settings.py` and edit them there. This will overload the previously imported defaults.

> Warning: It may be tempting to copy everything from `settings_default.py` into your own settings file. There is a reason we don't do this out of the box though: it makes it directly clear what changes you did. Also, if you limit your copying to the things you really need you will directly be able to take advantage of upstream changes and additions to Evennia for anything you didn't customize.

In code, the settings is accessed through

```
from django.conf import settings
 # or (shorter):
from evennia import settings
 # example:
servername = settings.SERVER_NAME
```

Each setting appears as a property on the imported `settings` object. You can also explore all possible options with `evennia.settings_full` (this also includes advanced Django defaults that are not touched in default Evennia).

> It should be pointed out that when importing `settings` into your code like this, it will be *read only*. You *cannot* edit your settings from your code! The only way to change an Evennia setting is to edit `mygame/server/conf/settings.py` directly. You also generally need to restart the server (possibly also the Portal) before a changed setting becomes available.

Other files in the `server/conf` directory

Apart from the main `settings.py` file,

- `at_initial_setup.py` - this allows you to add a custom startup method to be called (only) the very first time Evennia starts (at the same time as user #1 and Limbo is created). It can be made to start your own global scripts or set up other system/world-related things your game needs to have running from the start.
- `at_server_startstop.py` - this module contains two functions that Evennia will call every time the Server starts and stops respectively - this includes stopping due to reloading and resetting as well as shutting down completely. It's a useful place to put custom startup code for handlers and other things that must run in your game but which has no database persistence.
- `connection_screens.py` - all global string variables in this module are interpreted by Evennia as a greeting screen to show when a Player first connects. If more than one string variable is present in the module a random one will be picked.
- `inlinefuncs.py` - this is where you can define custom Inline functions.
- `inputfuncs.py` - this is where you define custom Input functions to handle data from the client.

- `lockfuncs.py` - this is one of many possible modules to hold your own "safe" *lock functions* to make available to Evennia's Locks.
- `mssp.py` - this holds meta information about your game. It is used by MUD search engines (which you often have to register with) in order to display what kind of game you are running along with statistics such as number of online players and online status.
- `oobfuncs.py` - in here you can define custom OOB functions.
- `portal_services_plugin.py` - this allows for adding your own custom services/protocols to the Portal. It must define one particular function that will be called by Evennia at startup. There can be any number of service plugin modules, all will be imported and used if defined. More info can be found here.
- `server_services_plugin.py` - this is equivalent to the previous one, but used for adding new services to the Server instead. More info can be found here.

Some other Evennia systems can be customized by plugin modules but has no explicit template in `conf/`:

- *commandparser.py* - a custom module can be used to totally replace Evennia's default command parser. All this does is to split the incoming string into "command name" and "the rest". It also handles things like error messages for no-matches and multiple-matches among other things that makes this more complex than it sounds. The default parser is *very* generic, so you are most often best served by modifying things further down the line (on the command parse level) than here.
- *at_search.py* - this allows for replacing the way Evennia handles search results. It allows to change how errors are echoed and how multi-matches are resolved and reported (like how the default understands that "2-ball" should match the second "ball" object if there are two of them in the room).

ServerConf

There is a special database model called `ServerConf` that stores server internal data and settings such as current player count (for interfacing with the webserver), startup status and many other things. It's rarely of use outside the server core itself but may be good to
know about if you are an Evennia developer.

Portal And Server

Evennia consists of two processes, known as *Portal* and *Server*. They can be controlled from inside the game or from the command line as described here.

If you are new to the concept, the main purpose of separating the two is to have players connect to the Portal but keep the MUD running on the Server. This way one can restart/reload the game (the Server part) without Players getting disconnected.

The Server and Portal are glued together via an AMP (Asynchronous Messaging Protocol) connection. This allows the two programs to communicate seamlessly.

Sessions

An Evennia *Session* represents one single established connection to the server. Depending on the Evennia session, it is possible for a person to connect multiple times, for example using different clients in multiple windows. Each such connection is represented by a session object.

A session object has its own cmdset, usually the "unloggedin" cmdset. This is what is used to show the login screen and to handle commands to create a new account (or Player in evennia lingo) read initial help and to log into the game with an existing account.

> Warning: A Session is not *persistent* - it is not a typeclass and has no connection to the database. The Session will go away when a user disconnects and you will lose any custom data on it if the server reloads. The `.db` handler on Sessions is there to present a uniform API, but if you were to read its doc you'll find it's actually just an alias to `.ndb`. So don't store any data on Sessions that you can't afford to lose in a reload. You have been warned.

A session object can either be "logged in" or not. Logged in means that the user has authenticated. When this happens the session is associated with a Player object (which is what holds account-centric stuff). The player can then in turn puppet any number of objects/characters.

Multisession mode

The number of sessions possible to connect to a given player at the same time and how it works is given by the `MULTISESSION_MODE` setting:

- `MULTISESSION_MODE=0`: One session per player. When connecting with a new session the old one is disconneted. This is the default mode and emulates many classic mud code bases.
- `MULTISESSION_MODE=1`: Many sessions per player, input from each session is treated identically. This means that you could use any number of different clients to input something and all would see the same result back.
- `MULTISESSION_MODE=2`: Many sessions per player, one character per session. This is the multi-playing mode where each session may, through one player account, individually puppet its own object/character without affecting what happens in other sessions.
- `MULTIPLAYER_MODE=3`: Many sessions per player *and* character. This is the full multi-puppeting mode, where multiple sessions may not only connect to the player account but multiple sessions may also puppet a single character at the same time. Tt is a multi-session version of mode 1. This allows multiplaying of many characters from any number of clients at once.

Returning data to the session

When you use `msg()` to return data to a user, the object on which you call the `msg()` matters. The `MULTISESSION_MODE` also matters, especially if greater than 1.

For example, if you use `player.msg("hello")` there is no way for evennia to know which session it should send the greeting to. In this case it will send it to all sessions. If you want a specific session you need to supply its session id (`sessid`) to the `msg` call.

On the other hand, if you call the `msg()` message on a puppeted object, like `character.msg("hello")`, the character already knows the sessid of the session that controls it - it will cleverly auto-add this for you (you can specify a different sessid if you specifically want to send stuff to another session).

Finally, there is a wrapper for `msg()` on all command classes: `command.msg()`. This will transparently detect which session was triggering the command (if any) and redirects to that session (this is most often what you want). If you are having trouble redirecting to a given session, `command.msg()` is often the safest bet.

You can get the `sessid` in many ways.

- A session stores the sessid in a property `sessid`.
- The player's `sessions` property holds all sessions connected to this player.
- Puppeted objects (normally Characters) have the persistent `sessid` property of the session puppeting them.

- Commands store the `sessid` pointing back to the session that triggered them (will be `None` if no session is involved, like when a mob or script triggers the command).

Customizing the session object

When would one want to customize the Session object? Consider for example a character creation system: You might decide to keep this on the out-of-character level. This would mean that you create the character at the end of some sort of menu choice. The actual char-create cmdset would then normally be put on the player. This works fine as long as you are `MULTISESSION_MODE` below 2. For higher modes, replacing the Player cmdset will affect *all* your connected sessions, also those not involved in character creation. In this case you want to instead put the char-create cmdset on the Session level - then all other sessions will keep working normally despite you creating a new character in one of them.

By default, the session object gets the `commands.default_cmdsets.UnloggedinCmdSet` when the user first connects. Once the session is authenticated it has *no* default sets. To add a "logged-in" cmdset to the Session, give the path to the cmdset class with `settings.CMDSET_SESSION`. This set will then henceforth always be present as soon as the player logs in.

To customize further you can completely override the Session with your own subclass. To replace the default Session class, change `settings.SERVER_SESSION_CLASS` to point to your custom class. This is a dangerous practice and errors can easily make your game unplayable. Make sure to take heed of the original and make your changes carefully.

Portal and Server Sessions

Note: This is considered an advanced topic. You don't need to know this on a first read-through.

Evennia is split into two parts, the Portal and the Server. Each side tracks its own Sessions, syncing them to each other.

The "Session" we normally refer to is actually the `ServerSession`. Its counter-part on the Portal side is the `PortalSession`. Whereas the server sessions deal with game states, the portal session deals with details of the connection-protocol itself. The two are also acting as backups of critical data such as when the server reboots.

New Player connections are listened for and handled by the Portal using the protocols it understands (such as telnet, ssh, webclient etc). When a new connection is established, a `PortalSession` is created on the Portal side. This session object looks different depending on which protocol is used to connect, but all still have a minimum set of attributes that are generic to all
sessions.

These common properties are piped from the Portal, through the AMP connection, to the Server, which is now informed a new connection has been established. On the Server side, a `ServerSession` object is created to represent this. There is only one type of `ServerSession`; It looks the same regardless of how the Player connects.

From now on, there is a one-to-one match between the `ServerSession` on one side of the AMP connection and the `PortalSession` on the other. Data arriving to the Portal Session is sent on to its mirror Server session and vice versa.

During certain situations, the portal- and server-side sessions are
"synced" with each other:

- The Player closes their client, killing the Portal Session. The Portal syncs with the Server to make sure the corresponding Server Session is also deleted.
- The Player quits from inside the game, killing the Server Session. The Server then syncs with the Portal to make sure to close the Portal connection cleanly.
- The Server is rebooted/reset/shutdown - The Server Sessions are copied over ("saved") to the Portal side. When the Server comes back up, this data is returned by the Portal so the two are again in sync. This way a Player's login status and other connection-critical things can survive a server reboot (assuming the Portal is not stopped at the same time, obviously).

Sessionhandlers

Both the Portal and Server each have a *sessionhandler* to manage the connections. These handlers contain all methods for relaying data across the AMP bridge. All types of Sessions hold a reference to their respective Sessionhandler (the property is called `sessionhandler`) so they can relay data. See protocols for more info on building new protocols.

Commands

Commands are intimately linked to Command Sets and you need to read that page too to be familiar with how the command system works. The two pages were split for easy reading.

The basic way for users to communicate with the game is through *Commands*. These can be commands directly related to the game world such as *look*, *get*, *drop* and so on, or administrative commands such as *examine* or *@dig*.

The default commands coming with Evennia are 'MUX-like' in that they use @ for admin commands, support things like switches, syntax with the '=' symbol etc, but there is nothing that prevents you from implementing a completely different command scheme for your game. You can find the default commands in `evennia/commands/default`. You should not edit these directly - they will be updated by the Evennia team as new features are added. Rather you should look to them for inspiration and inherit your own designs from them.

There are two components to having a command running - the *Command* class and the Command Set (command sets were split into a separate wiki page for ease of reading).

1. A *Command* is a python class containing all the functioning code for what a command does - for example, a *get* command would contain code for picking up objects.
2. A *Command Set* (often referred to as a CmdSet or cmdset) is like a container for one or more Commands. A given Command can go into any number of different command sets. Only by putting the command set on a character object you will make all the commands therein available to use by that character. You can also store command sets on normal objects if you want users to be able to use the object in various ways. Consider a "Tree" object with a cmdset defining the commands *climb* and *chop down*. Or a "Clock" with a cmdset containing the single command *check time*.

This page goes into full detail about how to use Commands. To fully use them you must also read the page detailing Command Sets. There is also a step-by-step Adding Command Tutorial that will get you started quickly without the extra explanations.

Defining Commands

All commands are implemented as normal Python classes inheriting from the base class `Command` (`evennia.Command`). You will find that this base class is very "bare". The default commands of Evennia actually inherit from

a child of `Command` called `MuxCommand` - this is the class that knows all the mux-like syntax like `/switches`, splitting by "=" etc. Below we'll avoid mux-specifics and use the base `Command` class directly.

```
# basic Command definition
from evennia import Command
class MyCmd(Command):
    """
    This is the help-text for the command
    """
    key = "mycommand"
    def parse(self):
        # parsing the command line here
    def func(self):
        # executing the command here
```

You define a new command by assigning a few class-global properties on your inherited class and overloading one or two hook functions. The full gritty mechanic behind how commands work are found towards the end of this page; for now you only need to know that the command handler creates an instance of this class and uses that instance whenever you use this command - it also dynamically assigns the new command instance a few useful properties that you can assume to always be available.

Who is calling the command?

In Evennia there are three types of objects that may call the command. It is important to be aware of this since this will also assign appropriate `caller`, `session`, `sessid` and `player` properties on the command body at runtime. Most often the calling type is `Session`.

- A Session. This is by far the most common case when a user is entering a command in their client.
  - `caller` - this is set to the puppeted Object if such an object exists. If no puppet is found, `caller` is set equal to `player`. Only if a Player is not found either (such as before being logged in) will this be set to the Session object itself.
  - `session` - a reference to the Session object itself.
  - `sessid` - `sessid.id`, a unique integer identifier of the session.
  - `player` - the Player object connected to this Session. None if not logged in.
- A Player. This only happens if `player.execute_cmd()` was used. No Session information can be obtained in this case.
  - `caller` - this is set to the puppeted Object if such an object can be determined (without Session info this can only be determined in `MULTISESSION_MODE=0` or `1`). If no puppet is found, this is equal to `player`.
  - `session` - None
  - `sessid` - None
  - `player` - Set to the Player object.
- An Object. This only happens if `object.execute_cmd()` was used (for example by an NPC).
  - `caller` - This is set to the calling Object in question.
  - `session` - None
  - `sessid` - None
  - `player` - None

Properties assigned to the command instance at run-time

Let's say player *Bob* with a character *BigGuy* enters the command *look at sword*. After the system having successfully identified this as the "look" command and determined that BigGuy really has access to a command named `look`, it chugs the `look` command class out of storage and either loads an existing Command instance from cache or creates one. After some more checks it then assigns it the following properties:

- `caller` - The character BigGuy, in this example. This is a reference to the object executing the command. The value of this depends on what type of object is calling the command; see the previous section.
- `session` - the Session Bob uses to connect to the game and control BigGuy (see also previous section).
- `sessid` - the unique id of `self.session`, for quick lookup.
- `player` - the Player Bob (see previous section).
- `cmdstring` - the matched key for the command. This would be *look* in our example.
- `args` - this is the rest of the string, except the command name. So if the string entered was *look at sword*, `args` would be " *at sword*". Note the space kept - Evennia would correctly interpret `lookat sword` too. This is useful for things like `/switches` that should not use space. In the `MuxCommand` class used for default commands, this space is stripped. Also see the `arg_regex` property if you want to enforce a space to make `lookat sword` give a command-not-found error.
- `obj` - the game Object on which this command is defined. This need not be the caller, but since `look` is a common (default) command, this is probably defined directly on *BigGuy* - so `obj` will point to BigGuy. Otherwise `obj` could be a Player or any interactive object with commands defined on it, like in the example of the "check time" command defined on a "Clock" object.
- `cmdset` - this is a reference to the merged CmdSet (see below) from which this command was matched. This variable is rarely used, it's main use is for the auto-help system (*Advanced note: the merged cmdset need NOT be the same as ''BigGuy.cmdset''. The merged set can be a combination of the cmdsets from other objects in the room, for example*).
- `raw_string` - this is the raw input coming from the user, without stripping any surrounding whitespace. The only thing that is stripped is the ending newline marker.

Defining your own command classes

Beyond the properties Evennia always assigns to the command at run-time (listed above), your job is to define the following class properties:

- `key` (string) - the identifier for the command, like `look`. This should (ideally) be unique. A key can consist of more than one word, like "press button" or "pull left lever". Note that *both* `key` and `aliases` below determine the identity of a command. So two commands are considered if either matches. This is important for merging cmdsets described below.
- `aliases` (optional list) - a list of alternate names for the command (`["glance", "see", "l"]`). Same name rules as for `key` applies.
- `locks` (string) - a lock definition, usually on the form `cmd:<lockfuncs>`. Locks is a rather big topic, so until you learn more about locks, stick to giving the lockstring `"cmd:all()"` to make the command available to everyone (if you don't provide a lock string, this will be assigned for you).
- `help_category` (optional string) - setting this helps to structure the auto-help into categories. If none is set, this will be set to *General*.
- `save_for_next` (optional boolean). This defaults to `False`. If `True`, a copy of this command object (along with any changes you have done to it) will be stored by the system and can be accessed by the next command by retrieving `self.caller.ndb.last_cmd`. The next run command will either clear or replace the storage.

- `arg_regex` (optional raw string): This should be given as a raw regular expression string. The regex will be compiled by the system at runtime. This allows you to customize how the part *immediately following* the command name (or alias) must look in order for the parser to match for this command. Normally the parser is highly efficient in picking out the command name, also as the beginning of a longer word (as long as the longer word is not a command name in it self). So `"lookme"` will be parsed as the command `"look"` followed by the argument `"me"`. By using `arg_regex` you could for example force the parser to require the command name to be followed by a space and then some other argument (regex `r"\s.+"`). Or you could allow both that and a stand-alone command (regex `r"\s.+|$"`). In this case, `look` and `"look me"` will work whereas `"lookme"` will lead to a "command not found" error.
- `auto_help` (optional boolean). Defaults to `True`. This allows for turning off the auto-help system on a per-command basis. This could be useful if you either want to write your help entries manually or hide the existence of a command from `help`'s generated list.
- `is_exit` (bool) - this marks the command as being used for an in-game exit. This is, by default, set by all Exit objects and you should not need to set it manually unless you make your own Exit system. It is used for optimization and allows the cmdhandler to easily disregard this command when the cmdset has its `no_exits` flag set.
- `is_channel` (bool)- this marks the command as being used for an in-game channel. This is, by default, set by all Channel objects and you should not need to set it manually unless you make your own Channel system. is used for optimization and allows the cmdhandler to easily disregard this command when its cmdset has its `no_channels` flag set.
- `msg_all_sessions` (bool): This affects the behavior of the `Command.msg` method. If unset (default), calling `self.msg(text)` from the Command will always only send text to the Session that actually triggered this Command. If set however, `self.msg(text)` will send to all Sessions relevant to the object this Command sits on. Just which Sessions receives the text depends on the object and the server's `MULTISESSION_MODE`.

You should also implement at least two methods, `parse()` and `func()` (You could also implement `perm()`, but that's not needed unless you want to fundamentally change how access checks work).

- `at_pre_cmd()` is called very first on the command. If this function returns anything that evaluates to `True` the command execution is aborted at this point.
- `parse()` is intended to parse the arguments (`self.args`) of the function. You can do this in any way you like, then store the result(s) in variable(s) on the command object itself (i.e. on `self`). To take an example, the default mux-like system uses this method to detect "command switches" and store them as a list in `self.switches`. Since the parsing is usually quite similar inside a command scheme you should make `parse()` as generic as possible and then inherit from it rather than re-implementing it over and over. In this way, the default `MuxCommand` class implements a `parse()` for all child commands to use.
- `func()` is called right after `parse()` and should make use of the pre-parsed input to actually do whatever the command is supposed to do. This is the main body of the command. The return value from this method will be returned from the execution as a Twisted Deferred.
- `at_post_cmd()` is called after `func()` to handle eventual cleanup.

Finally, you should always make an informative doc string (`__doc__`) at the top of your class. This string is dynamically read by the Help System to create the help entry for this command. You should decide on a way to format your help and stick to that.

Below is how you define a simple alternative "`smile`" command:

```
from evennia import Command

class CmdSmile(Command):
    """
    A smile command
```

```
    Usage:
      smile [at] [<someone>]
      grin [at] [<someone>]

    Smiles to someone in your vicinity or to the room
    in general.

    (This initial string (the __doc__ string)
    is also used to auto-generate the help
    for this command)
    """

    key = "smile"
    aliases = ["smile at", "grin", "grin at"]
    locks = "cmd:all()"
    help_category = "General"

    def parse(self):
        "Very trivial parser"
        self.target = self.args.strip()

    def func(self):
        "This actually does things"
        caller = self.caller
        if not self.target or self.target == "here":
            string = "%s smiles." % caller.name
            caller.location.msg_contents(string, exclude=caller)
            caller.msg("You smile.")
        else:
            target = caller.search(self.target)
            if not target:
                # caller.search handles error messages
                return
            string = "%s smiles to you." % caller.name
            target.msg(string)
            string = "You smile to %s." % target.name
            caller.msg(string)
            string = "%s smiles to %s." % (caller.name, target.name)
            caller.location.msg_contents(string, exclude=[caller,target])
```

The power of having commands as classes and to separate `parse()` and `func()` lies in the ability to inherit functionality without having to parse every command individually. For example, as mentioned the default commands all inherit from `MuxCommand`. `MuxCommand` implements its own version of `parse()` that understands all the specifics of MUX-like commands. Almost none of the default commands thus need to implement `parse()` at all, but can assume the incoming string is already split up and parsed in suitable ways by its parent.

Before you can actually use the command in your game, you must now store it within a *command set*. See the Command Sets page.

System commands

Note: This is an advanced topic. Skip it if this is your first time learning about commands.

There are several command-situations that are exceptional in the eyes of the server. What happens if the player enters an empty string? What if the 'command' given is infact the name of a channel the user wants to send a message to? Or if there are multiple command possibilities?

Such 'special cases' are handled by what's called *system commands*. A system command is defined in the same way as other commands, except that their name (key) must be set to one reserved by the engine (the names are defined at the top of `evennia/commands/cmdhandler.py`). You can find (unused) implementations of the system commands in `evennia/commands/default/system_commands.py`. Since these are not (by default) included in any `CmdSet` they are not actually used, they are just there for show. When the special situation occurs, Evennia will look through all valid `CmdSet`s for your custom system command. Only after that will it resort to its own, hard-coded implementation.

Here are the exceptional situations that triggers system commands. You can find the command keys they use as properties on `evennia.syscmdkeys`:

- No input (`syscmdkeys.CMD_NOINPUT`) - the player just pressed return without any input. Default is to do nothing, but it can be useful to do something here for certain implementations such as line editors that interpret non-commands as text input (an empty line in the editing buffer).
- Command not found (`syscmdkeys.CMD_NOMATCH`) - No matching command was found. Default is to display the "Huh?" error message.
- Several matching commands where found (`syscmdkeys.CMD_MULTIMATCH`) - Default is to show a list of matches.
- User is not allowed to execute the command (`syscmdkeys.CMD_NOPERM`) - Default is to display the "Huh?" error message.
- Channel (`syscmdkeys.CMD_CHANNEL`) - This is a Channel name of a channel you are subscribing to - Default is to relay the command's argument to that channel. Such commands are created by the Comm system on the fly depending on your subscriptions.
- New session connection (`syscmdkeys.CMD_LOGINSTART`). This command name should be put in the `settings.CMDSET_UNLOGGEDIN`. Whenever a new connection is established, this command is always called on the server (default is to show the login screen).

Below is an example of redefining what happens when the player doesn't provide any input (e.g. just presses return). Of course the new system command must be added to a cmdset as well before it will work.

```
from evennia import syscmdkeys, Command

class MyNoInputCommand(Command):
    "Usage: Just press return, I dare you"
    key = syscmdkeys.CMD_NOINPUT
    def func(self):
        self.caller.msg("Don't just press return like that, talk to me!")
```

Dynamic Commands

Note: This is an advanced topic.

Normally Commands are created as fixed classes and used without modification. There are however situations when the exact key, alias or other properties is not possible (or impractical) to pre-code (Exits is an example of this).

To create a command with a dynamic call signature, first define the command body normally in a class (set your `key`, `aliases` to default values), then use the following call (assuming the command class you created is named `MyCommand`):

```
cmd = MyCommand(key="newname",
                aliases=["test", "test2"],
                locks="cmd:all()",
                ...)
```

All keyword arguments you give to the Command constructor will be stored as a property on the command object. This will overload existing properties defined on the parent class.

Normally you would define your class and only overload things like `key` and `aliases` at run-time. But you could in principle also send method objects (like `func`) as keyword arguments in order to make your command completely customized at run-time.

Exits

Note: This is an advanced topic.

Exits are examples of the use of a Dynamic Command.

The functionality of Exit objects in Evennia is not hard-coded in the engine. Instead Exits are normal typeclassed objects that auto-create a CmdSet on themselves when they load. This cmdset has a single dynamically created Command with the same properties (key, aliases and locks) as the Exit object itself. When entering the name of the exit, this dynamic exit-command is triggered and (after access checks) moves the Character to the exit's destination.

Whereas you could customize the Exit object and its command to achieve completely different behaviour, you will usually be fine just using the appropriate `traverse_*` hooks on the Exit object. But if you are interested in really changing how things work under the hood, check out `evennia/objects/objects.py` for how the `Exit` typeclass is set up.

How commands actually work

Note: This is an advanced topic mainly of interest to server developers.

Any time the user sends text to Evennia, the server tries to figure out if the text entered corresponds to a known command. This is how the command handler sequence looks for a logged-in user:

1. A user enters a string of text and presses enter.

- The user's Session determines the text is not some protocol-specific control sequence or OOB command, but sends it on to the command handler.
- Evennia's *command handler* analyzes the Session and grabs eventual references to Player and eventual puppeted Characters (these will be stored on the command object later). The *caller* property is set appropriately.
- If input is an empty string, resend command as `CMD_NOINPUT`. If no such command is found in cmdset, ignore.
- If command.key matches `settings.IDLE_COMMAND`, update timers but don't do anything more.

1. The command handler gathers the CmdSets available to *caller* at this time:

- The caller's own currently active CmdSet.
- CmdSets defined on the current player, if caller is a puppeted object.
- CmdSets defined on the Session itself.
- The active CmdSets of eventual objects in the same location (if any). This includes commands on Exits.
- Sets of dynamically created *System commands* representing available Communications.

1. All !CmdSets *of the same priority* are merged together in groups. Grouping avoids order-dependent issues of merging multiple same-prio sets onto lower ones.
2. All the grouped CmdSets are *merged* in reverse priority into one combined CmdSet according to each set's merge rules.

3. Evennia's *command parser* takes the merged cmdset and matches each of its commands (using its key and aliases) against the beginning of the string entered by *caller*. This produces a set of candidates.
4. The *cmd parser* next rates the matches by how many characters they have and how many percent matches the respective known command. Only if candidates cannot be separated will it return multiple matches.

- If multiple matches were returned, resend as `CMD_MULTIMATCH`. If no such command is found in cmdset, return hard-coded list of matches.
- If no match was found, resend as `CMD_NOMATCH`. If no such command is found in cmdset, give hard-coded error message.

1. If a single command was found by the parser, the correct command object is plucked out of storage. This usually doesn't mean a re-initialization.
2. It is checked that the caller actually has access to the command by validating the *lockstring* of the command. If not, it is not considered as a suitable match and `CMD_NOMATCH` is triggered.
3. If the new command is tagged as a channel-command, resend as `CMD_CHANNEL`. If no such command is found in cmdset, use hard-coded implementation.
4. Assign several useful variables to the command instance (see previous sections).
5. Call `at_pre_command()` on the command instance.
6. Call `parse()` on the command instance. This is fed the remainder of the string, after the name of the command. It's intended to pre-parse the string into a form useful for the `func()` method.
7. Call `func()` on the command instance. This is the functional body of the command, actually doing useful things.
8. Call `at_post_command()` on the command instance.

Assorted notes

The return value of `Command.func()` is a Twisted deferred. Evennia does not use this return value at all by default. If you do, you must thus do so asynchronously, using callbacks.

```
# in command class func()
def callback(ret, caller):
   caller.msg("Returned is %s" % ret)
deferred = self.execute_command("longrunning")
deferred.addCallback(callback, self.caller)
```

This is probably not relevant to any but the most advanced/exotic designs (one might use it to create a "nested" command structure for example).

The `save_for_next` class variable can be used to implement state-persistent commands. For example it can make a command operate on "it", where it is determined by what the previous command operated on.

Command Sets

Command Sets are intimately linked with Commands and you should be familiar with Commands before reading this page. The two pages were split for ease of reading.

A *Command Set* (often referred to as a CmdSet or cmdset) is the basic unit for storing one or more *Commands*. A given Command can go into any number of different command sets. Storing Command classes in a command set is the *only* way to make the commands available to use in your game.

When storing a CmdSet on an object, will you make the commands in that command set available to the object. An example is the default command set stored on new Characters. This command set contains all the useful commands, from `look` and `inventory` to `@dig` and `@reload` (permissions then limit which players may use them, but that's a separate topic).

When a player enters a command, cmdsets from the Player, Character, its location and elsewhere are pulled together into a *merge stack*. This stack is merged together in a specific order to create a single "merged" cmdset, representing the pool of commands available at that very moment.

An example would be a `Window` object that has a cmdset with two commands in it: `look through window` and `open window`. The command set would be visible to players in the room with the window, allowing them to use those commands only there. You could imagine all sorts of clever uses of this, like a `Television` object which had multiple commands for looking at it, switching channels and so on. The tutorial world coming with Evennia showcases a dark room that replaces certain critical commands with its on versions because the Character cannot see.

If you want a quick start into defining your first commands and use them with command sets, you can head over to the Adding Command Tutorial which steps through things without the explanations.

Defining Command Sets

A CmdSet is, as most things in Evennia, defined as a Python class inheriting from the correct parent (`evennia.CmdSet`, which is a shortcut to `evennia.commands.cmdset.CmdSet`). The CmdSet class only needs to define one method, called `at_cmdset_creation()`. All other class parameters are optional, but are used for more advanced set manipulation and coding (see the *merge rules* section).

```
# file mygame/commands/mycmdset.py

from evennia import CmdSet

# this is a theoretical custom module with commands we
# created previously: mygame/commands/mycommands.py
from commands import mycommands

class MyCmdSet(CmdSet):
    def at_cmdset_creation(self):
        """
        The only thing this method should need
        to do is to add commands to the set.
        """
        self.add(mycommands.MyCommand1())
        self.add(mycommands.MyCommand2())
        self.add(mycommands.MyCommand3())
```

The CmdSet's `add()` method can also take another CmdSet as input. In this case all the commands from that CmdSet will be appended to this one as if you added them line by line:

```
def at_cmdset_creation():
    ...
    self.add(AdditionalCmdSet) # adds all command from this set
    ...
```

If you added your command to an existing cmdset (like to the default cmdset), that set is already loaded into memory. You need to make the server aware of the code changes:

```
@reload
```

You should now be able to use the command.

If you created a new, fresh cmdset, this must be added to an object in order to make the commands within available. A simple way to temporarily test a cmdset on yourself is use the `@py` command to execute a python snippet:

```
@py self.cmdset.add('commands.mycmdset.MyCmdSet')
```

This will stay with you until you `@reset` or `@shutdown` the server, or you run

```
@py self.cmdset.delete('commands.mycmdset.MyCmdSet')
```

Above a specific Cmdset class is removed. Calling `delete` without arguments will remove the latest added cmdset.

> Note: Command sets added using `cmdset.add` are by default *not* persistent in the database.

If you want the cmdset to survive a reload, you can do

```
@py self.cmdset.add(commands.mycmdset.MyCmdSet, permanent=True)
```

or you could add the cmdset as the *default* cmdset:

```
@py self.cmdset.add_default(commands.mycmdset.MyCmdSet)
```

An object can only have one "default" cmdset (but can also have none). This is meant as a safe fall-back even if all other cmdsets fail or are removed. It is always persistent and will not be affected by `cmdset.delete()`. To remove a default cmdset you must explicitly call `cmdset.remove_default()`.

Command sets are often added to an object in its `at_object_creation` method. For more examples of adding commands, read the Step by step tutorial. Generally you can customize which command sets are added to your objects by using `self.cmdset.add()` or `self.cmdset.add_default()`.

> Important: Commands are identified uniquely by key *or* alias (see Commands). If any overlap exists, two commands are considered identical. Adding a Command to a command set that already has an identical command will *replace* the previous command. This is very important in order to easily overload default Evennia commands with your own, but you need be aware of this or you may accidentally "hide" your own command in your command set because you add a new one that happen to have a matching alias.

Properties on command sets

There are a few extra flags one can set on CmdSets in order to modify how they work. All are optional and will be set to defaults otherwise. Since many of these relate to *merging* cmdsets you might want to read up on next section for some of these to make sense.

- `key` (string) - an identifier for the cmdset. This is optional but should be unique; it is used for display in lists but also to identify special merging behaviours using the `key_mergetype` dictionary below. - `mergetype` (string) - one of "*Union*", "*Intersect*", "*Replace*" or "*Remove*".
- `priority` (int) - This defines the merge order of the merge stack - cmdsets will merge in rising order of priority with the highest priority set merging last. During a merger, the commands from the set with the higher priority will have precedence (just what happens depends on the *merge type*). If priority is identical, the order in the merge stack determines preference. The priority value must be greater or equal to `-100`. Most in-game sets should usually have priorities between `0` and `100`. Evennia default sets have priorities as follows (these can be changed if you want a different distribution):
  - EmptySet: `-101` (should be lower than all other sets)
  - SessionCmdSet: `-20`
  - PlayerCmdSet: `-10`
  - CharacterCmdSet: `0`

  - ExitCmdSet: `101` (generally should always be available)
  - ChannelCmdSet: `101` (should usually always be available) - since exits never accept arguments, there is no collision between exits named the same as a channel even though the commands "collide".

- `key_mergetype` (dict) - a dict of `key:mergetype` pairs. This allows this cmdset to merge differently with certain named cmdsets. If the cmdset to merge with has a `key` matching an entry in `key_mergetype`, it will not be merged according to the setting in `mergetype` but according to the mode in this dict. Please note that this is more complex than it may seem due to the *merge order* of command sets. Please review that section before using `key_mergetype`.
- `duplicates` (bool/None default `None`) - this determines what happens when merging same-priority cmdsets containing same-key commands together. The`dupicate` option will *only* apply when merging the cmdset with this option onto one other cmdset with the same priority. The resulting cmdset will *not* retain this `duplicate` setting.
  - `None` (default): No duplicates are allowed and the cmdset being merged "onto" the old one will take precedence. The result will be unique commands. *However*, the system will assume this value to be `True` for cmdsets on Objects, to avoid dangerous clashes. This is usually the safe bet.
  - `False`: Like `None` except the system will not auto-assume any value for cmdsets defined on Objects.
  - `True`: Same-named, same-prio commands will merge into the same cmdset. This will lead to a multimatch error (the user will get a list of possibilities in order to specify which command they meant). This is is useful e.g. for on-object cmdsets (example: There is a `red button` and a `green button` in the room. Both have a `press button` command, in cmdsets with the same priority. This flag makes sure that just writing `press button` will force the Player to define just which object's command was intended).
- `no_objs` this is a flag for the cmdhandler that builds the set of commands available at every moment. It tells the handler not to include cmdsets from objects around the player (nor from rooms or inventory) when building the merged set. Exit commands will still be included. This option can have three values:
  - `None` (default): Passthrough of any value set explicitly earlier in the merge stack. If never set explicitly, this acts as `False`.
  - `True`/`False`: Explicitly turn on/off. If two sets with explicit `no_objs` are merged, priority determines what is used.
- `no_exits` - this is a flag for the cmdhandler that builds the set of commands available at every moment. It tells the handler not to include cmdsets from exits. This flag can have three values:
  - `None` (default): Passthrough of any value set explicitly earlier in the merge stack. If never set explicitly, this acts as `False`.
  - `True`/`False`: Explicitly turn on/off. If two sets with explicit `no_exits` are merged, priority determines what is used.
- `no_channels` (bool) - this is a flag for the cmdhandler that builds the set of commands available at every moment. It tells the handler not to include cmdsets from available in-game channels. This flag can have three values:
  - `None` (default): Passthrough of any value set explicitly earlier in the merge stack. If never set explicitly, this acts as `False`.
  - `True`/`False`: Explicitly turn on/off. If two sets with explicit `no_channels` are merged, priority determines what is used.

Command Sets searched

When a user issues a command, it is matched against the *merged* command sets available to the player at the moment. Which those are may change at any time (such as when the player walks into the room with the `Window` object

described earlier).

The currently valid command sets are collected from the following sources:

- The cmdsets stored on the currently active Session. Default is the empty `SessionCmdSet` with merge priority `-20`.
- The cmdsets defined on the Player. Default is the PlayerCmdSet with merge priority `-10`.
- All cmdsets on the Character/Object (assuming the Player is currently puppeting such a Character/Object). Merge priority `0`.
- The cmdsets of all objects carried by the puppeted Character (checks the `call` lock). Will not be included if `no_objs` option is active in the merge stack.
- The cmdsets of the Character's current location (checks the `call` lock). Will not be included if `no_objs` option is active in the merge stack.
- The cmdsets of objects in the current location (checks the `call` lock). Will not be included if `no_objs` option is active in the merge stack.
- The cmdsets of Exits in the location. Merge priority `+101`. Will not be included if `no_exits` *or* `no_objs` option is active in the merge stack.
- The channel cmdset containing commands for posting to all channels the player or character is currently connected to. Merge priority `+101`. Will not be included if `no_channels` option is active in the merge stack.

Note that an object does not *have* to share its commands with its surroundings. A Character's cmdsets should not be shared for example, or all other Characters would get multi-match errors just by being in the same room. The ability of an object to share its cmdsets is managed by its `call` lock. For example, Character objects defaults to `call:false()` so that any cmdsets on them can only be accessed by themselves, not by other objects around them. Another example might be to lock an object with `call:inside()` to only make their commands available to objects inside them, or `cmd:holds()` to make their commands available only if they are held.

Adding and merging command sets

Note: This is an advanced topic. It's very useful to know about, but you might want to skip it if this is your first time learning about commands.

CmdSets have the special ability that they can be *merged* together into new sets. Which of the ingoing commands end up in the merged set is defined by the *merge rule* and the relative *priorities* of the two sets. Removing the latest added set will restore things back to the way it was before the addition.

CmdSets are non-destructively stored in a stack inside the cmdset handler on the object. This stack is parsed to create the "combined" cmdset active at the moment. CmdSets from other sources are also included in the merger such as those on objects in the same room (like buttons to press) or those introduced by state changes (such as when entering a menu). The cmdsets are all ordered after priority and then merged together in *reverse order*. That is, the higher priority will be merged "onto" lower-prio ones. By defining a cmdset with a merge-priority between that of two other sets, you will make sure it will be merged in between them.

The very first cmdset in this stack is called the *Default cmdset* and is protected from accidental deletion. Running `obj.cmdset.delete()` will never delete the default set. Instead one should add new cmdsets on top of the default to "hide" it, as described below. Use the special `obj.cmdset.delete_default()` only if you really know what you are doing.

CmdSet merging is an advanced feature useful for implementing powerful game effects. Imagine for example a player entering a dark room. You don't want the player to be able to find everything in the room at a glance - maybe you even

want them to have a hard time to find stuff in their backpack! You can then define a different CmdSet with commands that override the normal ones. While they are in the dark room, maybe the `look` and `inv` commands now just tell the player they cannot see anything! Another example would be to offer special combat commands only when the player is in combat. Or when being on a boat. Or when having taken the super power-up. All this can be done on the fly by merging command sets.

Merge rules

Basic rule is that command sets are merged in *reverse priority order*. That is, lower-prio sets are merged first and higher prio sets are merged "on top" of them. Think of it like a layered cake with the highest priority on top.

To further understand how sets merge, we need to define some examples. Let's call the first command set **A** and the second **B**. We assume **B** is the command set already active on our object and we will merge **A** onto **B**. In code terms this would be done by `object.cdmset.add(A)`. Remember, B is already active on `object` from before.

We let the **A** set have higher priority than **B**. A priority is simply an integer number. As seen in the list above, Evennia's default cmdsets have priorities in the range `-101` to `120`. You are usually safe to use a priority of `0` or `1` for most game effects.

In our examples, both sets contain a number of commands which we'll identify by numbers, like `A1, A2` for set **A** and `B1, B2, B3, B4` for **B**. So for that example both sets contain commands with the same keys (or aliases) "1" and "2" (this could for example be "look" and "get" in the real game), whereas commands 3 and 4 are unique to **B**. To describe a merge between these sets, we would write `A1,A2 + B1,B2,B3,B4 = ?` where `?` is a list of commands that depend on which merge type **A** has, and which relative priorities the two sets have. By convention, we read this statement as "New command set **A** is merged onto the old command set **B** to form **?**".

Below are the available merge types and how they work. Names are partly borrowed from Set theory.

- **Union** (default) - The two cmdsets are merged so that as many commands as possible from each cmdset ends up in the merged cmdset. Same-key commands are merged by priority.

```
# Union
A1,A2 + B1,B2,B3,B4 = A1,A2,B3,B4
```

- **Intersect** - Only commands found in *both* cmdsets (i.e. which have the same keys) end up in the merged cmdset, with the higher-priority cmdset replacing the lower one's commands.

```
# Intersect
A1,A3,A5 + B1,B2,B4,B5 = A1,A5
```

- **Replace** - The commands of the higher-prio cmdset completely replaces the lower-priority cmdset's commands, regardless of if same-key commands exist or not.

```
# Replace
A1,A3 + B1,B2,B4,B5 = A1,A3
```

- **Remove** - The high-priority command sets removes same-key commands from the lower-priority cmdset. They are not replaced with anything, so this is a sort of filter that prunes the low-prio set using the high-prio one as a template.

```
# Remove
A1,A3 + B1,B2,B3,B4,B5 = B2,B4,B5
```

Besides `priority` and `mergetype`, a command-set also takes a few other variables to control how they merge:

- `duplicates` (bool) - determines what happens when two sets of equal priority merge. Default is that the new set in the merger (i.e. **A** above) automatically takes precedence. But if *duplicates* is true, the result will be a merger with more than one of each name match. This will usually lead to the player receiving a multiple-match

error higher up the road, but can be good for things like cmdsets on non-player objects in a room, to allow the system to warn that more than one 'ball' in the room has the same 'kick' command defined on it and offer a chance to select which ball to kick ... Allowing duplicates only makes sense for *Union* and *Intersect*, the setting is ignored for the other mergetypes.

- `key_mergetypes` (dict) - allows the cmdset to define a unique mergetype for particular cmdsets, identified by their cmdset `key`. Format is `{CmdSetkey:mergetype}`. Example: `{'Myevilcmdset','Replace'}` which would make sure for this set to always use 'Replace' on the cmdset with the key `Myevilcmdset` only, no matter what the main `mergetype` is set to.

  Warning: The `key_mergetypes` dictionary *can only work on the cmdset we merge onto*. When using `key_mergetypes` it is thus important to consider the merge priorities - you must make sure that you pick a priority *between* the cmdset you want to detect and the next higher one, if any. That is, if we define a cmdset with a high priority and set it to affect a cmdset that is far down in the merge stack, we would not "see" that set when it's time for us to merge. Example: Merge stack is `A(prio=-10), B(prio=-5), C(prio=0), D(prio=5)`. We now merge a cmdset `E(prio=10)` onto this stack, with a `key_mergetype={"B":"Replace"}`. But priorities dictate that we won't be merged onto B, we will be merged onto E (which is a merger of the lower-prio sets at this point). Since we are merging onto E and not B, our `key_mergetype` directive won't trigger. To make sure it works we must make sure we merge onto B. Setting E's priority to, say, -4 will make sure to merge it onto B and affect it appropriately.

More advanced cmdset example:

```
from commands import mycommands

class MyCmdSet(CmdSet):

    key = "MyCmdSet"
    priority = 4
    mergetype = "Replace"
    key_mergetypes = {'MyOtherCmdSet':'Union'}

    def at_cmdset_creation(self):
        """
        The only thing this method should need
        to do is to add commands to the set.
        """
        self.add(mycommands.MyCommand1())
        self.add(mycommands.MyCommand2())
        self.add(mycommands.MyCommand3())
```

Assorted notes

It is very important to remember that two commands are compared *both* by their `key` properties *and* by their `aliases` properties. If either keys or one of their aliases match, the two commands are considered the *same*. So consider these two Commands:

- A Command with key "kick" and alias "fight"
- A Command with key "punch" also with an alias "fight"

During the cmdset merging (which happens all the time since also things like channel commands and exits are merged in), these two commands will be considered *identical* since they share alias. It means only one of them will remain after the merger. Each will also be compared with all other commands having any combination of the keys and/or aliases "kick", "punch" or "fight".

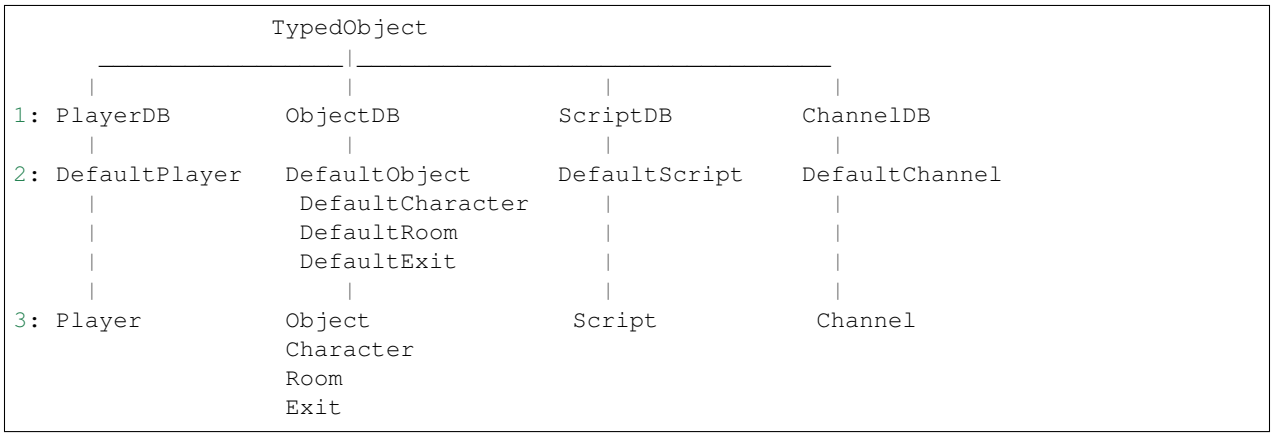
... So avoid duplicate aliases, it will only cause confusion.

Typeclasses

Typeclasses form the core of Evennia data storage. It allows Evennia to represent any number of different game entities as Python classes, without having to modify the database schema for every new type.

In Evennia the most important game entities, Players, Objects, Scripts and Channels are all Python classes inheriting, at varying distance, from `evennia.typeclasses.models.TypedObject`. In the documentation we refer to these objects as being "typeclassed" or even "being a typeclass".

This is how the inheritance looks for the typeclasses in Evennia:



Level 1 above is the "database model" level. This describes the database tables and fields (this is technically a Django model).

Level 2 is where we find Evennia's default implementations of the various game entities, on top of the database. These classes define all the hook methods that Evennia calls in various situations. `DefaultObject` is a little special since it's the parent for `DefaultCharacter`, `DefaultRoom` and `DefaultExit`. They are all grouped under level 2 because they all represents defaults to build from.

Level 3, finally, holds empty template classes created in your game directory. This is the level you are meant to modify and tweak as you please, overloading the defaults as befits your game. The templates inherit directly from their defaults, so `Object` inherits from `DefaultObject` and `Room` inherits from `DefaultRoom`.

Difference between typeclasses and classes

All Evennia classes inheriting from class in the table above share one important feature and two important limitations. This is why we don't simply call them "classes" but "typeclasses".

1. A typeclass can save itself to the database. This means that some properties (actually not that many) on the class actually represents database fields and can only hold very specific data types. This is detailed below.
2. Due to its connection to the database, the typeclass' name must be *unique* across the *entire* server namespace. That is, there must never be two same-named classes defined anywhere. So the below code would give an error (since `DefaultObject` is now globally found both in this module and in the default library):

```
from evennia import DefaultObject as BaseObject
class DefaultObject(BaseObject):
    pass
```

3. A typeclass' `__init__` method should normally not be overloaded. This has mostly to do with the fact that the `__init__` method is not called in a predictable way. Instead Evennia suggest you use the `at_*_creation` hooks (like `at_object_creation` for Objects) for setting things the very first time the typeclass is saved to

the database or the `at_init` hook which is called every time the object is cached to memory. If you know what you are doing and want to use `__init__`, it *must* both accept arbitrary keyword arguments and use `super` to call its parent:

```
.. code:: python
```

def __init__(self, **kwargs): # my content super(MyTypeclass, self).__init__(**kwargs) # my content

Apart from this, a typeclass works like any normal Python class and you can treat it as such.

Creating a new typeclass

It's easy to work with Typeclasses. Either you use an existing typeclass or you create a new Python class inheriting from an existing typeclass. Here is an example of creating a new type of Object:

‘‘‘python
from evennia import DefaultObject

```
class Furniture(DefaultObject):
    # this defines what 'furniture' is, like
    # storing who sits on it or something.
    pass
```

You can now create a new `Furniture` object in two ways. First (and usually not the most convenient) way is to create an instance of the class and then save it manually to the database:

```
chair = Furniture(db_key="Chair")
chair.save()
```

To use this you must give the database field names as keywords to the call. Which are available depends on the entity you are creating, but all start with `db_*` in Evennia. This is a method you may be familiar with if you know Django from before.

It is recommended that you instead use the `create_*` functions to create typeclassed entities:

```
from evennia import create_object

chair = create_object(Furniture, key="Chair")
# or (if your typeclass is in a module furniture.py)
chair = create_object("furniture.Furniture", key="Chair")
```

The `create_object` (`create_player`, `create_script` etc) takes the typeclass as its first argument; this can both be the actual class or the python path to the typeclass as found under your game directory. So if your `Furniture` typeclass sits in `mygame/typeclasses/furniture.py`, you could point to it as `typeclasses.furniture.Furniture`. Since Evennia will itself look in `mygame/typeclasses`, you can shorten this even further to just `furniture.Furniture`. The create-functions take a lot of extra keywords allowing you to set things like Attributes and Tags all in one go. These keywords don't use the `db_*` prefix. This will also automatically save the new instance to the database, so you don't need to call `save()` explicitly.

About typeclass properties

An example of a database field is `db_key`. This stores the "name" of the entity you are modifying and can thus only hold a string. This is one way of making sure to update the `db_key`:

```
chair.db_key = "Table"
chair.save()

print chair.db_key
<<< Table
```

That is, we change the chair object to have the `db_key` "Table", then save this to the database. However, you almost never do things this way; Evennia defines property wrappers for all the database fields. These are named the same as the field, but without the `db_` part:

```
chair.key = "Table"

print chair.key
<<< Table
```

The `key` wrapper is not only shorter to write, it will make sure to save the field for you, and does so more efficiently by levering sql update mechanics under the hood. So whereas it is good to be aware that the field is named `db_key` you should use `key` as much as you can.

Each typeclass entity has some unique fields relevant to that type. But all also share the following fields (the wrapper name without `db_` is given):

- `key` (str): The main identifier for the entity, like "Rose", "myscript" or "Paul". `name` is an alias.
- `date_created` (datetime): Time stamp when this object was created.
- `typeclass_path` (str): A python path pointing to the location of this (type)class

There is one special field that doesn't use the `db_` prefix (it's defined by Django):

- `id` (int): the database id (database ref) of the object. This is an ever-increasing, unique integer. It can also be accessed as `dbid` (database ID) or `pk` (primary key). The `dbref` property returns the string form "#id".

The typeclassed entity has several common handlers:

- `tags` - the TagHandler that handles tagging. Use `tags.add()` , `tags.get()` etc.
- `locks` - the LockHandler that manages access restrictions. Use `locks.add()`, `locks.get()` etc.
- `attributes` - the AttributeHandler that manages Attributes on the object. Use `attributes.add()` etc.
- `db` (DataBase) - a shortcut property to the AttributeHandler; allowing `obj.db.attrname = value`
- `nattributes` - the Non-persistent AttributeHandler for attributes not saved in the database.
- `ndb` (NotDataBase) - a shortcut property to the Non-peristent AttributeHandler. Allows `obj.ndb.attrname = value`

Each of the typeclassed entities then extend this list with their own properties. Go to the respective pages for Objects, Scripts, Players and Channels for more info. It's also recommended that you explore the available entities using Evennia's flat API to explore which properties and methods they have available.

Overloading hooks

The way to customize typeclasses is usually to overload *hook methods* on them. Hooks are methods that Evennia call in various situations. An example is the `at_object_creation` hook on `Objects`, which is only called once,

the very first time this object is saved to the database. Other examples are the `at_login` hook of Players and the `at_repeat` hook of Scripts.

Querying for typeclasses

Most of the time you search for objects in the database by using convenience methods like the `caller.search()` of Commands or the search functions like `evennia.search_objects`.

You can however also query for them directly using Django's query language. This makes use of a *database manager* that sits on all typeclasses, named `objects`. This manager holds methods that allow database searches against that particular type of object (this is the way Django normally works too). When using Django queries, you need to use the full field names (like `db_key`) to search:

```
matches = Furniture.objects.get(db_key="Chair")
```

It is important that this will *only* find objects inheriting directly from `Furniture` in your database. If there was a subclass of `Furniture` named `Sitables` you would not find any chairs derived from `Sitables` with this query (this is not a Django feature but special to Evennia). To find objects from subclasses Evennia instead makes the `get_family` and `filter_family` query methods available:

```
# search for all furnitures and subclasses of furnitures
# whose names starts with "Chair"
matches = Furniture.objects.filter_family(db_key__startswith="Chair")
```

To make sure to search, say, all `Scripts` *regardless* of typeclass, you need to query from the database model itself. So for Objects, this would be `ObjectDB` in the diagram above. Here's an example for Scripts:

```
from evennia import ScriptDB
matches = ScriptDB.objects.filter(db_key__contains="Combat")
```

When querying from the database model parent you don't need to use `filter_family` or `get_family` - you will always query all children on the database model.

Updating existing typeclass instances

If you already have created instances of Typeclasses, you can modify the *Python code* at any time - due to how Python inheritance works your changes will automatically be applied to all children once you have reloaded the server.

However, database-saved data, like `db_*` fields, Attributes, Tags etc, are not themselves embedded into the class and will *not* be updated automatically. This you need to manage yourself, by searching for all relevant objects and updating or adding the data:

```
# add a worth Attribute to all existing Furniture
for obj in Furniture.objects.all():
    # this will loop over all Furniture instances
    obj.db.worth = 100
```

A common use case is putting all Attributes in the `at_*_creation` hook of the entity, such as `at_object_creation` for `Objects`. This is called every time an object is created - and only then. This is usually what you want but it does mean already existing objects won't get updated if you change the contents of `at_object_creation` later. You can fix this in a similar way as above (manually setting each Attribute) or with something like this:

```
# Re-run at_object_creation only on those objects not having the new Attribute
for obj in Furniture.objects.all():
```

```
    if not obj.db.worth:
        obj.at_object_creation()
```

The above examples can be run in the command prompt created by `evennia shell`. You could also run it all in-game using `@py`. That however requires you to put the code (including imports) as one single line using `;` and list comprehensions, like this (ignore the line break, that's only for readability in the wiki):

```
@py from typeclasses.furniture import Furniture;
[obj.at_object_creation() for obj in Furniture.objects.all() if not obj.db.worth]
```

It is recommended that you plan your game properly before starting to build, to avoid having to retroactively update objects more than necessary.

Swap typeclass

If you want to swap an already existing typeclass, there are two ways to do so: From in-game and via code. From inside the game you can use the default `@typeclass` command:

```
@typeclass objname = path.to.new.typeclass
```

There are two important switches to this command:

- `/reset` - This will purge all existing Attributes on the object and re-run the creation hook (like `at_object_creation` for Objects). This assures you get an object which is purely of this new class.
- `/force` - This is required if you are changing the class to be *the same* class the object already has - it's a safety check to avoid user errors. This is usually used together with `/reset` to re-run the creation hook on an existing class.

In code you instead use the `swap_typeclass` method which you can find on all typeclassed entities:

```
obj_to_change.swap_typeclass(new_typeclass_path, clean_attributes=False,
                    run_start_hooks=True, no_default=True)
```

The arguments to this method are described in the API docs here.

How typeclasses actually work

This is considered an advanced section.

Technically, typeclasses are Django proxy models. The only database models that are "real" in the typeclass system (that is, are represented by actual tables in the database) are `PlayerDB`, `ObjectDB`, `ScriptDB` and `ChannelDB` (there are also Attributes and Tags but they are not typeclasses themselves). All the subclasses of them are "proxies", extending them with Python code without actually modifying the database layout.

Evennia modifies Django's proxy model in various ways to allow them to work without any boiler plate (for example you don't need to set the Django "proxy" property in the model `Meta` subclass, Evennia handles this for you using metaclasses). Evennia also makes sure you can query subclasses as well as patches django to allow multiple inheritance from the same base class.

Caveats

Evennia uses the *idmapper* to cache its typeclasses (Django proxy models) in memory. The idmapper allows things like on-object handlers and properties to be stored on typeclass instances and to not get lost as long as the server is running (they will only be cleared on a Server reload). Django does not work like this by default; by default every time

you search for an object in the database you'll get a *different* instance of that object back and anything you stored on it that was not in the database would be lost. The bottom line is that Evennia's Typeclass instances subside in memory a lot longer than vanilla Django model instance do.

There is one caveat to consider with this, and that relates to making your own models: Foreign relationships to typeclasses are cached by Django and that means that if you were to change an object in a foreign relationship via some other means than via that relationship, the object seeing the relationship may not reliably update but will still see its old cached version. Due to typeclasses staying so long in memory, stale caches of such relationships could be more visible than common in Django. See the closed issue #1098 and its comments for examples and solutions.

Objects

All in-game objects in Evennia, be it characters, chairs, monsters, rooms or hand grenades are represented by an Evennia *Object*. Objects form the core of Evennia and is probably what you'll spend most time working with. Objects are Typeclassed entities.

How to create your own object types

An Evennia Object is, per definition, a Python class that includes `evennia.DefaultObject` among its parents. In `mygame/typeclasses/objects.py` there is already a class `Object` that inherits from `DefaultObject` and that you can inherit from. You can put your new typeclass directly in that module or you could organize your code in some other way. Here we assume we make a new module `mygame/typeclasses/flowers.py`:

```
# mygame/typeclasses/flowers.py

from typeclasses.objects import Object

class Rose(Object):
    """
    This creates a simple rose object
    """
    def at_object_creation(self):
        "this is called only once, when object is first created"
        # add a persistent attribute 'desc'
        # to object (silly example).
        self.db.desc = "This is a pretty rose with thorns."
```

You could save this in the `mygame/typeclasses/objects.py` (then you'd not need to import `Object`) or you can put it in a new module. Let's say we do the latter, making a module `typeclasses/flowers.py`. Now you just need to point to the class *Rose* with the `@create` command to make a new rose:

```
@create/drop MyRose:flowers.Rose
```

What the `@create` command actually *does* is to use `evennia.create_object`. You can do the same thing yourself in code:

```
from evennia import create_object
new_rose = create_object("typeclasses.flowers.Rose", key="MyRose")
```

(The `@create` command will auto-append the most likely path to your typeclass, if you enter the call manually you have to give the full path to the class. The `create.create_object` function is powerful and should be used for all coded object creating (so this is what you use when defining your own building commands). Check out the `ev.create_*` functions for how to build other entities like Scripts).

This particular Rose class doesn't really do much, all it does it make sure the attribute `desc`(which is what the `look` command looks for) is pre-set, which is pretty pointless since you will usually want to change this at build time (using the `@desc` command or using the Spawner). The `Object` typeclass offers many more hooks that is available to use though - see next section.

Properties and functions on Objects

Beyond the properties assigned to all typeclassed objects (see that page for a list of those), the Object also has the following custom properties:

- `aliases` - a handler that allows you to add and remove aliases from this object. Use `aliases.add()` to add a new alias and `aliases.remove()` to remove one.
- `location` - a reference to the object currently containing this object.
- `home` is a backup location. The main motivation is to have a safe place to move the object to if its `location` is destroyed. All objects should usually have a home location for safety.
- `destination` - this holds a reference to another object this object links to in some way. Its main use is for Exits, it's otherwise usually unset.
- `nicks` - as opposed to aliases, a Nick holds a convenient nickname replacement for a real name, word or sequence, only valid for this object. This mainly makes sense if the Object is used as a game character - it can then store briefer shorts, example so as to quickly reference game commands or other characters.or nicks.add(alias, realname) to add a new one.
- `player` - this holds a reference to a connected Player controlling this object (if any). Note that this is set also if the controlling player is *not* currently online - to test if a player is online, use the `has_player` property instead.
- `sessions` - if `player` field is set *and the player is online*, this is a list of all active sessions (server connections) to contact them through (it may be more than one if multiple connections are allowed in settings).
- `has_player` - a shorthand for checking if an *online* player is currently connected to this object.
- `contents` - this returns a list referencing all objects 'inside' this object (i,e. which has this object set as their `location`).
- `exits` - this returns all objects inside this object that are *Exits*, that is, has the `destination` property set.

The last two properties are special:

- `cmdset` - this is a handler that stores all command sets defined on the object (if any).
- `scripts` - this is a handler that manages Scripts attached to the object (if any).

The Object also has a host of useful utility functions. See the function headers in `src/objects/objects.py` for their arguments and more details.

- `msg()` - this function is used to send messages from the server to a player connected to this object.
- `msg_contents()` - calls `msg` on all objects inside this object.
- `search()` - this is a convenient shorthand to search for a specific object, at a given location or globally. It's mainly useful when defining commands (in which case the object executing the command is named `caller` and one can do `caller.search()` to find objects in the room to operate on).
- `execute_cmd()` - Lets the object execute the given string as if it was given on the command line.
- `move_to` - perform a full move of this object to a new location. This is the main move method and will call all relevant hooks, do all checks etc.
- `clear_exits()` - will delete all Exits to *and* from this object.

- `clear_contents()` - this will not delete anything, but rather move all contents (except Exits) to their designated `Home` locations.
- `delete()` - deletes this object, first calling `clear_exits()` and `clear_contents()`.

The Object Typeclass defines many more *hook methods* beyond `at_object_creation`. Evennia calls these hooks at various points. When implementing your custom objects, you will inherit from the base parent and overload these hooks with your own custom code. See `evennia.objects.objects` for an updated list of all the available hooks or the API for DefaultObject here.

Subclasses of `Object`

There are three special subclasses of *Object* in default Evennia - *Characters*, *Rooms* and *Exits*. The reason they are separated is because these particular object types are fundamental, something you will always need and in some cases requires some extra attention in order to be recognized by the game engine (there is nothing stopping you from redefining them though). In practice they are all pretty similar to the base Object.

Characters

Characters are objects controlled by Players. When a new Player logs in to Evennia for the first time, a new `Character` object is created and the Player object is assigned to the `player` attribute. A `Character` object must have a Default Commandset set on itself at creation, or the player will not be able to issue any commands! If you just inherit your own class from `evennia.DefaultCharacter` and make sure to use `super()` to call the parent methods you should be fine. In `mygame/typeclasses/characters.py` is an empty `Character` class ready for you to modify.

Rooms

Rooms are the root containers of all other objects. The only thing really separating a room from any other object is that they have no `location` of their own and that default commands like `@dig` creates objects of this class - so if you want to expand your rooms with more functionality, just inherit from `ev.DefaultRoom`. In `mygame/typeclasses/rooms.py` is an empty `Room` class ready for you to modify.

Exits

Exits are objects connecting other objects (usually *Rooms*) together. An object named *North* or *in* might be an exit, as well as *door*, *portal* or *jump out the window*. An exit has two things that separate them from other objects. Firstly, their *destination* property is set and points to a valid object. This fact makes it easy and fast to locate exits in the database. Secondly, exits define a special Transit Command on themselves when they are created. This command is named the same as the exit object and will, when called, handle the practicalities of moving the character to the Exits's *destination* - this allows you to just enter the name of the exit on its own to move around, just as you would expect.

The exit functionality is all defined on the Exit typeclass, so you could in principle completely change how exits work in your game (it's not recommended though, unless you really know what you are doing). Exits are locked using an access_type called *traverse* and also make use of a few hook methods for giving feedback if the traversal fails. See `evennia.DefaultExit` for more info. In `mygame/typeclasses/exits.py` there is an empty `Exit` class for you to modify.

The process of traversing an exit is as follows:

1. The traversing `obj` sends a command that matches the Exit-command name on the Exit object. The cmdhandler detects this and triggers the command defined on the Exit. Traversal always involves the "source" (the current location) and the `destination` (this is stored on the Exit object).

2. The Exit command checks the `traverse` lock on the Exit object
3. The Exit command triggers `at_traverse(obj, destination)` on the Exit object.
4. In `at_traverse`, `object.move_to(destination)` is triggered. This triggers the following hooks, in order:
   (a) `obj.at_before_move(destination)` - if this returns False, move is aborted.
   (b) `origin.at_before_leave(obj, destination)`
   (c) `obj.announce_move_from(destination)`
   (d) Move is performed by changing `obj.location` from source location to `destination`.
   (e) `obj.announce_move_to(source)`
   (f) `destination.at_object_receive(obj, source)`
   (g) `obj.at_after_move(source)`
5. On the Exit object, `at_after_traverse(obj, source)` is triggered.

If the move fails for whatever reason, the Exit will look for an Attribute `err_traverse` on itself and display this as an error message. If this is not found, the Exit will instead call
`at_failed_traverse(obj)` on itself.

Scripts

Scripts are the out-of-character siblings to the in-character Objects. The name "Script" might suggest that they can only be used to script the game but this is only part of their usefulness (in the end we had to pick a single name for them). Scripts are persistent in the database just like Objects and you can attach Attributes to it.

Scripts can be used for many different things in Evennia:

- They can attach to Objects to influence them in various ways - or exist independently of any one in-game entity.
- They can work as timers and tickers - anything that may change with Time. But they can also have no time dependence at all.
- They can describe State changes.
- They can act as data stores for storing game data persistently in the database.
- They can be used as OOC stores for sharing data between groups of objects.

A Script is an excellent platform for hosting a persistent, but unique system handler. For example, a Script could be used as the base to track the state of a turn-based combat system. Since Scripts can also operate on a timer they can also update themselves regularly to perform various actions.

A Script is very powerful and together with its ability to hold Attributes you can use it for data storage in various ways. However, if all you want is just to have an object method called repeatedly, you should consider using the TickerHandler which is more limited but is specialized on just this task.

How to create and test your own Script types

In-game you can try out scripts using the `@script` command. Try the following to apply the script to your character.

```
> @script self = bodyfunctions.BodyFunctions
```

Or, if you want to inflict your flatulence script on another person, place or thing, try something like the following:

```
> @py self.location.search('matt').scripts.add('bodyfunctions.BodyFunctions')
```

This should cause some random messages to appear at random intervals. You can find this example in `evennia/contrib/tutorial_examples/bodyfunctions.py`.

```
> @script/stop self = bodyfunctions.BodyFunctions
```

This will kill the script again. You can use the `@scripts` command to list all active scripts in the game, if any (there are none by default).

If you add scripts to Objects the script can then manipulate the object as desired. The script is added to the object's *script handler*, called simply `scripts`. The handler takes care of all initialization and startup of the script for you.

```
# add script to myobj's scripthandler
myobj.scripts.add("myscripts.CoolScript")
# alternative way
from evennia import create_script
create_script("myscripts.CoolScript", obj=myobj)
```

A script does not have to be connected to an in-game object. If not it is called a *Global script*. You can create global scripts by simply not supplying an object to store it on:

```
# adding a global script
from evennia import create_script
create_script("typeclasses.globals.MyGlobalEconomy",
              key="economy", persistent=True, obj=None)
```

You can create a global script manually using `@py` or by putting the above for example in `mygame/server/conf/at_initial_startup.py`, which means the script will be created only once, when the server is started for the very first time (there are other files in the `mygame/server/conf/` folder that triggers at other times).

Properties and functions defined on Scripts

A Script has all the properties of a typeclassed object, such as `db` and `ndb`(see Typeclasses). Setting `key` is useful in order to manage scripts (delete them by name etc). These are usually set up in the Script's typeclass, but can also be assigned on the fly as keyword arguments to `evennia.create_script`.

- `desc` - an optional description of the script's function. Seen in script listings.
- `interval` - how often the script should run. If `interval == 0` (default), it runs forever, without any repeating (it will not accept a negative value).
- `start_delay` - (bool), if we should wait `interval` seconds before firing for the first time or not.
- `repeats` - How many times we should repeat, assuming `interval > 0`. If repeats is set to `<= 0`, the script will repeat indefinitely.
- `persistent`- if this script should survive a server *reset* or server *shutdown*. (You don't need to set this for it to survive a normal reload - the script will be paused and seamlessly restart after the reload is complete).

There is one special property:

- `obj` - the Object this script is attached to (if any). You should not need to set this manually. If you add the script to the Object with `myobj.scripts.add(myscriptpath)` or give `myobj` as an argument to the `utils.create.create_script` function, the `obj` property will be set to `myobj` for you.

It's also imperative to know the hook functions. Normally, overriding these are all the customization you'll need to do in Scripts. You can find longer descriptions of these in `src/scripts/scripts.py`.

- `at_script_creation()` - this is usually where the script class sets things like `interval` and `repeats`; things that control how the script runs. It is only called once - when the script is first created.
- `is_valid()` - determines if the script should still be running or not. This is called when running `obj.scripts.validate()`, which you can run manually, but which is also called by Evennia during certain situations such as reloads. This is also useful for using scripts as state managers. If the method returns `False`, the script is stopped and cleanly removed.
- `at_start()` - this is called when the script starts or is unpaused. For persistent scripts this is at least once ever server startup. Note that this will *always* be called right away, also if `start_delay` is `True`.
- `at_repeat()` - this is called every `interval` seconds, or not at all. It is called right away at startup, unless `start_delay` is `True`, in which case the system will wait `interval` seconds before calling.
- `at_stop()` - this is called when the script stops for whatever reason. It's a good place to do custom cleanup.
- `at_server_reload()` - this is called whenever the server is warm-rebooted (e.g. with the `@reload` command). It's a good place to save non-persistent data you might want to survive a reload.
- `at_server_shutdown()` - this is called when a system reset or systems shutdown is invoked.

Running methods (usually called automatically by the engine, but possible to also invoke manually)

- `start()` - this will start the script. This is called automatically whenever you add a new script to a handler. `at_start()` will be called.
- `stop()` - this will stop the script and delete it. Removing a script from a handler will stop it automatically. `at_stop()` will be called.
- `pause()` - this pauses a running script, rendering it inactive, but not deleting it. All properties are saved and timers can be resumed. This is called automatically when the server reloads and will *not* lead to the *at_stop()* hook being called. This is a suspension of the script, not a change of state.
- `unpause()` - resumes a previously paused script. The `at_start()` hook *will* be called to allow it to reclaim its internal state. Timers etc are restored to what they were before pause. The server automatically un-pauses all paused scripts after a server reload.
- `force_repeat()` - this will forcibly step the script, regardless of when it would otherwise have fired. The timer will reset and the `at_repeat()` hook is called as normal. This also counts towards the total number of repeats, if limited.
- `time_until_next_repeat()` - for timed scripts, this returns the time in seconds until it next fires. Returns `None` if `interval==0`.
- `remaining_repeats()` - if the Script should run a limited amount of times, this tells us how many are currently left.
- `reset_callcount(value=0)` - this allows you to reset the number of times the Script has fired. It only makes sense if `repeats > 0`.
- `restart(interval=None, repeats=None, start_delay=None)` - this method allows you to restart the Script in-place with different run settings. If you do, the `at_stop` hook will be called and the Script brought to a halt, then the `at_start` hook will be called as the Script starts up with your (possibly changed) settings. Any keyword left at `None` means to not change the original setting.

Defining new Scripts

There are two ways to create a new Script type:

1. Creating a Script instance using `evennia.create_script()`. This function takes all the important script parameters (`interval`, `locks`, `start_delay` etc) as keyword arguments. It will return a newly instanced (and started) script object. If you set the keyword `persistent=True`, the returned Script will survive a server reset/reboot too.
2. Define a new Script Typeclass. This you can do for example in the module `evennia/typeclasses/scripts.py`. Below is an example Script Typeclass.

```
import random
from evennia import DefaultScript

class Weather(DefaultScript):
    """Displays weather info. Meant to be attached to a room."""
    def at_script_creation(self):
        self.key = "weather_script"
        self.desc = "Gives random weather messages."
        self.interval = 60 * 5 # every 5 minutes
        self.persistent = True

    def at_repeat(self):
        "called every self.interval seconds."
        rand = random.random()
        if rand < 0.5:
            weather = "A faint breeze is felt."
        elif rand < 0.7:
            weather = "Clouds sweep across the sky."
        else:
            weather = "There is a light drizzle of rain."
        # send this message to everyone inside the object this
        # script is attached to (likely a room)
        self.obj.msg_contents(weather)
```

This is a simple weather script that we can put on an object. Every 5 minutes it will tell everyone inside that object how the weather is.

To activate it, just add it to the script handler (`scripts`) on an Room. That object becomes `self.obj` in the example above. Here we put it on a room called `myroom`:

```
myroom.scripts.add(weather.Weather)
```

Once you have the typeclass written you can feed your Typeclass to the `create_script` function directly:

```
from evennia import create_script
create_script('typeclasses.weather.Weather', obj=myroom)
```

Note that if you were to give a keyword argument to `create_script`, that would override the default value in your Typeclass. So for example:

```
create_script('typeclasses.weather.Weather', obj=myroom,
              persistent=False, interval=10*60)
```

This particular instance of the Weather Script would run with a 10 minute interval. It would also not survive a server reset/reboot.

From in-game you can use the `@script` command as usual to get to your Typeclass:

```
@script here = weather.Weather
```

You can conveniently view and kill running Scripts by using the `@scripts` command in-game.

Dealing with Errors

Errors inside an executing script can sometimes be rather terse or point to parts of the execution mechanism that is hard to interpret. One way to make it easier to debug scripts is to import Evennia's native logger and wrap your functions in a try/catch block. Evennia's logger can show you where the traceback occurred in your script.

```
from evennia.utils import logger

class Weather(DefaultScript):

    # [...]

    def at_repeat(self):

        try:
            # [...] code as above
        except Exception:
            # logs the error
            logger.log_trace()
```

More on Scripts

For another example of a Script in use, check out the Turn Based Combat System tutorial.

Players

All *gamers* (real people) that opens a game Session on Evennia are doing so through an object called *Player*. The Player object has no in-game representation, it represents a unique game account. In order to actually get on the game the Player must *puppet* an Object (normally a **'Character'_**).

Exactly how many Sessions can interact with a Player and its Puppets at once is determined by Evennia's MULTISESSION_MODE setting.

Apart from storing login information and other account-specific data, the Player object is what is chatting on Channels. It is also a good place to store Permissions to be consistent between different in-game characters as well as configuration options. The Player object also has its own CmdSet, the `PlayerCmdSet`.

logged into default evennia, you can use the `@ooc` command to leave your current **'character'_** and go into ooc mode. you are quite limited in this mode, basically it works like a simple chat program. It acts as a staging area for switching between Characters (if your game supports that) or as a safety mode if your Character gets deleted. Use `@ic` to attempt to puppet a Character.

Note that the Player object can and often do have a different set of Permissions from the Character they control. Normally you should put your permissions on the Player level - this will overrule permissions set on the Character level (unless `@quell`-ing is used).

How to create your own Player types

You will usually not want more than one Player typeclass for all new players (but you could in principle create a system that changes a player's typeclass dynamically).

An Evennia Player is, per definition, a Python class that includes `evennia.DefaultPlayer` among its parents. In `mygame/typeclasses/players.py` there is an empty class ready for you to modify. Evennia defaults to using this (it inherits directly from `DefaultPlayer`).

Here's an example of modifying the default Player class in code:

```
# in mygame/typeclasses/players.py

from evennia import DefaultPlayer

class Player(DefaultPlayer):
    # [...]

    at_player_creation(self):
        "this is called only once, when player is first created"
        self.db.real_name = None      # this is set later
        self.db.real_address = None   #       "
        self.db.config_1 = True       # default config
        self.db.config_2 = False      #       "
        self.db.config_3 = 1          #       "

        # ... whatever else our game needs to know
```

Reload the server with `@reload`.

... However, if you use `examine *self` (the asterisk makes you examine your Player object rather than your Character), you won't see your new Attributes yet. This is because `at_player_creation` is only called the very *first* time the Player is called and your Player object already exists (any new Players that connect will see them though). To update yourself you need to make sure to re-fire the hook on all the Players you have already created. Here is an example of how to do this using `@py`:

```
@py [player.at_player_creation() for player in evennia.managers.players.all()]
```

You should now see the Attributes on yourself.

> If you wanted Evennia to default to a completely *different* Player class located elsewhere, you must point Evennia to it. Add `BASE_PLAYER_TYPECLASS` to your settings file, and give the python path to your custom class as its value. By default this points to `typeclasses.players.Player`, the empty template we used above.

Properties on Players

Beyond those properties assigned to all typeclassed objects (see Typeclasses), the Player also has the following custom properties:

- `user` - a unique link to a `User` Django object, representing the logged-in user.
- `obj` - an alias for `character`.
- `name` - an alias for `user.username`
- `sessions` - a list of all connected Sessions (physical connections) this object listens to. The so-called session-id (used in many places) is found as a property `sessid` on each Session instance.

- `is_superuser` (bool: True/False) - if this player is a superuser.

Special handlers:

- `cmdset` - This holds all the current Commands of this Player. By default these are the commands found in the cmdset defined by `settings.CMDSET_PLAYER`.
- `nicks` - This stores and handles Nicks, in the same way as nicks it works on Objects. For Players, nicks are primarily used to store custom aliases for Channels.

Selection of special methods (see `evennia.DefaultPlayer` for details):

- `get_puppet` - get a currently puppeted object connected to the Player and a given session id, if any.
- `puppet_object` - connect a session to a puppetable Object.
- `unpuppet_object` - disconnect a session from a puppetable Object.
- `msg` - send text to the Player
- `execute_cmd` - runs a command as if this Player did it.
- `search` - search for Players.

Communications

Apart from moving around in the game world and talking, players might need other forms of communication. This is offered by Evennia's `Comm` system. Stock evennia implements a 'MUX-like' system of channels, but there is nothing stopping you from changing things to better suit your taste.

Comms rely on two main database objects - `Msg` and `Channel`. There is also the `TempMsg` which mimics the API of a `Msg` but has no connection to the database.

Msg

The `Msg` object is the basic unit of communication in Evennia. A message works a little like an e-mail; it always has a sender (a Player) and one or more recipients. The recipients may be either other Players, or a *Channel* (see below). You can mix recipients to send the message to both Channels and Players if you like.

Once created, a `Msg` is normally not changed. It is peristently saved in the database. This allows for comprehensive logging of communications. This could be useful for allowing senders/receivers to have 'mailboxes' with the messages they want to keep.

Properties defined on `Msg`

- `senders` - this is a reference to one or many Player or Objects (normally *Characters*) sending the message. This could also be an *External Connection* such as a message coming in over IRC/IMC2 (see below). There is usually only one sender, but the types can also be mixed in any combination.
- `receivers` - a list of target Players, Objects (usually *Characters*) or *Channels* to send the message to. The types of receivers can be mixed in any combination.
- `header` - this has a max-length of 128 characters. This could be used to store mime-type information for this type of message (such as if it's a mail or a page), but depending on your game it could also instead be used for the subject line or other types of header info you want to track. Being an indexed field it can be used for quick look-ups in the database.
- `message` - the actual text being sent.

- `date_sent` - when message was sent (auto-created).
- `locks` - a lock definition.
- `hide_from` - this can optionally hold a list of objects, players or channels to hide this `Msg` from. This relationship is stored in the database primarily for optimization reasons, allowing for quickly post-filter out messages not intended for a given target. There is no in-game methods for setting this, it's intended to be done in code.

You create new messages in code using `evennia.create_message` (or `evennia.utils.create.create_message`.)

TempMsg

`evennia.comms.models` also has `TempMsg` which mimics the API of `Msg` but is not connected to the database. TempMsgs are used by Evennia for channel messages by default. They can be used for any system expecting a `Msg` but when you don't actually want to save anything.

Channels

Channels are Typeclassed entities, which mean they can be easily extended and their functionality modified. To change which channel typeclass Evennia uses, change settings.BASE_CHANNEL_TYPECLASS.

Channels act as generic distributors of messages. Think of them as "switch boards" redistributing `Msg` or `TempMsg` objects. Internally they hold a list of "listening" objects and any `Msg` (or `TempMsg`) sent to the channel will be distributed out to all channel listeners. Channels have Locks to limit who may listen and/or send messages through them.

The *sending* of text to a channel is handled by a dynamically created Command that always have the same name as the channel. This is created for each channel by the global `ChannelHandler`. The Channel command is added to the Player's cmdset and normal command locks are used to determine which channels are possible to write to. When subscribing to a channel, you can then just write the channel name and the text to send.

The default ChannelCommand (which can be customized by pointing `settings.CHANNEL_COMMAND_CLASS` to your own command), implements a few convenient features:

- It only sends `TempMsg` objects. Instead of storing individual entries in the database it instead dumps channel output a file log in `server/logs/channel_<channelname>.log`. This is mainly for practical reasons - we find one rarely need to query individual Msg objects at a later date. Just stupidly dumping the log to a file also means a lot less database overhead.
- It adds a `/history` switch to view the 20 last messages in the channel. These are read from the end of the log file. One can also supply a line number to start further back in the file (but always 20 entries at a time). It's used like this:

```
> public/history
> public/history 35
```

There are two default channels created in stock Evennia - `MudInfo` and `Public`. `MudInfo` receives server-related messages meant for Admins whereas `Public` is open to everyone to chat on (all new players are automatically joined to it when logging in, it is useful for asking questions). The default channels are defined by the `DEFAULT_CHANNELS` list (see `evennia/settings_default.py` for more details).

You create new channels with `evennia.create_channel` (or `evennia.utils.create.create_channel`).

In code, messages are sent to a channel using the `msg` or `tempmsg` methods of channels:

```
channel.msg(msgobj, header=None, senders=None, persistent=True)
```

The argument `msgobj` can be either a string, a previously constructed `Msg` or a `TempMsg` - in the latter cases all the following keywords are ignored since the message objects already contains all this information. If `msgobj` is a string, the other keywords are used for creating a new `Msg` or `TempMsg` on the fly, depending on if `persistent` is set or not. By default, a `TempMsg` is emitted for channel communication (since the default ChannelCommand instead logs to a file).

```
# assume we have a 'sender' object and a channel named 'mychan'

# manually sending a message to a channel
mychan.msg("Hello!", senders=[sender])
```

Properties defined on `Channel`

- `key` - main name for channel
- `aliases` - alternative native names for channels
- `desc` - optional description of channel (seen in listings)
- `keep_log` (bool) - if the channel should store messages (default)
- `locks` - A lock definition. Channels normally use the access_types `send`, `admin` and `listen`.

Attributes

When performing actions in Evennia it is often important that you store data for later. If you write a menu system, you have to keep track of the current location in the menu tree so that the player can give correct subsequent commands. If you are writing a combat system, you might have a combattant's next roll get easier dependent on if their opponent failed. Your characters will probably need to store roleplaying-attributes like strength and agility. And so on.

Typeclassed game entities (Players, Objects, Scripts and Channels) always have *Attributes* associated with them. Attributes are used to store any type of data 'on' such entities. This is different from storing data in properties already defined on entities (such as `key` or `location`) - these have very specific names and require very specific types of data (for example you couldn't assign a python *list* to the `key` property no matter how hard you tried). `Attributes` come into play when you want to assign arbitrary data to arbitrary names.

The .db and .ndb shortcuts

To save persistent data on a Typeclassed object you normally use the `db` (DataBase) operator. Let's try to save some data to a *Rose* (an Object):

```
# saving
rose.db.has_thorns = True
# getting it back
is_ouch = rose.db.has_thorns
```

This looks like any normal Python assignment, but that `db` makes sure that an *Attribute* is created behind the scenes and is stored in the database. Your rose will continue to have thorns throughout the life of the server now, until you deliberately remove them.

To be sure to save **non-persistently**, i.e. to make sure NOT to create a database entry, you use `ndb` (NonDataBase). It works in the same way:

```
# saving
rose.ndb.has_thorns = True
# getting it back
is_ouch = rose.ndb.has_thorns
```

Technically, ndb has nothing to do with Attributes, despite how similar they look. No Attribute object is created behind the scenes when using ndb. In fact the database is not invoked at all since we are not interested in persistence. There is however an important reason to use ndb to store data rather than to just store variables direct on entities - ndb-stored data is tracked by the server and will not be purged in various cache-cleanup operations Evennia may do while it runs. Data stored on ndb (as well as db) will also be easily listed by example the @examine command.

You can also del properties on db and ndb as normal. This will for example delete an Attribute:

```
del rose.db.has_thorns
```

Both db and ndb defaults to offering an all() method on themselves. This returns all associated attributes or non-persistent properties.

```
list_of_all_rose_attributes = rose.db.all()
list_of_all_rose_ndb_attrs = rose.ndb.all()
```

If you use all as the name of an attribute, this will be used instead. Later deleting your custom all will return the default behaviour.

The AttributeHandler

The .db and .ndb properties are very convenient but if you don't know the name of the Attribute beforehand they cannot be used. Behind the scenes .db actually accesses the AttributeHandler which sits on typeclassed entities as the .attributes property. .ndb does the same for the .nattributes property.

The handlers have normal access methods that allow you to manage and retrieve Attributes and NAttributes:

- has(...) - this checks if the object has an Attribute with this key. This is equivalent to doing obj.db.key.
- get(...) - this retrieves the given Attribute. Normally the value property of the Attribute is returned, but the method takes keywords for returning the Attribute object itself. By supplying an accessing_object to the call one can also make sure to check permissions before modifying anything.
- add(...) - this adds a new Attribute to the object. An optional lockstring can be supplied here to restrict future access and also the call itself may be checked against locks.
- remove(...) - Remove the given Attribute. This can optionally be made to check for permission before performing the deletion. - clear(...) - removes all Attributes from object.
- all(...) - returns all Attributes (of the given category) attached to this object.

See this section for more about locking down Attribute access and editing. The Nattribute offers no concept of access control.

Some examples:

```
import evennia
obj = evennia.search_object("MyObject")

obj.attributes.add("test", "testvalue")
print obj.db.test                  # prints "testvalue"
print obj.attributes.get("test")   #        "
```

```
print obj.attributes.all()          # prints [<AttributeObject>]
obj.attributes.remove("test")
```

Properties of Attributes

An Attribute object is stored in the database. It has the following properties:

- `key` - the name of the Attribute. When doing e.g. `obj.db.attrname = value`, this property is set to `attrname`.
- `value` - this is the value of the Attribute. This value can be anything which can be pickled - objects, lists, numbers or what have you (see this section for more info). In the example `obj.db.attrname = value`, the `value` is stored here.
- `category` - this is an optional property that is set to None for most Attributes. Setting this allows to use Attributes for different functionality. This is usually not needed unless you want to use Attributes for very different functionality (Nicks is an example of using Attributes in this way). To modify this property you need to use the Attribute Handler.
- `strvalue` - this is a separate value field that only accepts strings. This severely limits the data possible to store, but allows for easier database lookups. This property is usually not used except when re-using Attributes for some other purpose (Nicks use it). It is only accessible via the Attribute Handler.

There are also two special properties:

- `attrtype` - this is used internally by Evennia to separate Nicks, from Attributes (Nicks use Attributes behind the scenes).
- `model` - this is a *natural-key* describing the model this Attribute is attached to. This is on the form *app-name.modelclass*, like `objects.objectdb`. It is used by the Attribute and NickHandler to quickly sort matches in the database. Neither this nor `attrtype` should normally need to be modified.

Non-database attributes have no equivalence to `category` nor `strvalue`, `attrtype` or `model`.

Persistent vs non-persistent

So *persistent* data means that your data will survive a server reboot, whereas with *non-persistent* data it will not ...

... So why would you ever want to use non-persistent data? The answer is, you don't have to. Most of the time you really want to save as much as you possibly can. Non-persistent data is potentially useful in a few situations though.

- You are worried about database performance. Since Evennia caches Attributes very aggressively, this is not an issue unless you are reading *and* writing to your Attribute very often (like many times per second). Reading from an already cached Attribute is as fast as reading any Python property. But even then this is not likely something to worry about: Apart from Evennia's own caching, modern database systems themselves also cache data very efficiently for speed. Our default database even runs completely in RAM if possible, alleviating much of the need to write to disk during heavy loads.
- A more valid reason for using non-persistent data is if you *want* to lose your state when logging off. Maybe you are storing throw-away data that are re-initialized at server startup. Maybe you are implementing some caching of your own. Or maybe you are testing a buggy Script that does potentially harmful stuff to your character object. With non-persistent storage you can be sure that whatever is messed up, it's nothing a server reboot can't clear up.
- NAttributes have no restrictions at all on what they can store (see next section), since they don't need to worry about being saved to the database - they work very well for temporary storage.
- You want to implement a fully or partly *non-persistent world*. Who are we to argue with your grand vision!

What types of data can I save in an Attribute?

> None of the following affects NAttributes, which does not invoke the database at all. There are no restrictions to what can be stored in a NAttribute.

The database doesn't know anything about Python objects, so Evennia must *serialize* Attribute values into a string representation in order to store it to the database. This is done using the `pickle` module of Python (the only exception is if you use the `strattr` keyword of the AttributeHandler to save to the `strvalue` field of the Attribute. In that case you can only save *strings* which will not be pickled).

Storing single objects

With a single object, we mean anything that is *not iterable*, like numbers, strings or custom class instances without the `__iter__` method.

- You can generally store any non-iterable Python entity that can be pickled.
- Single database objects/typeclasses can be stored as any other in the Attribute. These can normally *not* be pickled, but Evennia will behind the scenes convert them to an internal representation using their classname, database-id and creation-date with a microsecond precision, guaranteeing you get the same object back when you access the Attribute later.
- If you *hide* a database object inside a non-iterable custom class (like stored as a variable inside it), Evennia will not know it's there and won't convert it safely. Storing classes with such hidden database objects is *not* supported and will lead to errors!

```
# Examples of valid single-value  attribute data:
obj.db.test1 = 23
obj.db.test1 = False
# a database object (will be stored as an internal representation)
obj.db.test2 = myobj

# example of an invalid, "hidden" dbobject
class Invalid(object):
    def __init__(self, dbobj):
        # no way for Evennia to know this is a dbobj
        self.dbobj = dbobj
invalid = Invalid(myobj)
obj.db.invalid = invalid # will cause error!
```

Storing multiple objects

This means storing objects in a collection of some kind and are examples of *iterables*, pickle-able entities you can loop over in a for-loop. Attribute-saving supports the following iterables:

- Tuples, like `(1,2,"test", <dbobj>)`.
- Lists, like `[1,2,"test", <dbobj>]`.
- Dicts, like `{1:2, "test":<dbobj>]`.
- Sets, like `{1,2,"test",<dbobj>}`.
- collections.OdereredDict, like `OrderendDict((1,2), ("test", <dbobj>))`.
- collections.Deque, like `deque((1,2,"test",<dbobj>))`.
- *Nestings* of any combinations of the above, like lists in dicts or an OrderedDict of tuples, each containing dicts, etc.

- All other iterables (i.e. entities with the `__iter__` method) will be converted to a *list*. Since you can use any combination of the above iterables, this is generally not much of a limitation.

Any entity listed in the Single object section above can be stored in the iterable.

> As mentioned in the previous section, database entities (aka typeclasses) are not possible to pickle. So when storing an iterable, Evennia must recursively traverse the iterable *and all its nested sub-iterables* in order to find eventual database objects to convert. This is a very fast process but for efficiency you may want to avoid too deeply nested structures if you can.

```
# examples of valid iterables to store
obj.db.test3 = [obj1, 45, obj2, 67]
# a dictionary
obj.db.test4 = {'str':34, 'dex':56, 'agi':22, 'int':77}
# a mixed dictionary/list
obj.db.test5 = {'members': [obj1,obj2,obj3], 'enemies':[obj4,obj5]}
# a tuple with a list in it
obj.db.test6 = (1,3,4,8, ["test", "test2"], 9)
# a set will still be stored and returned as a list [1,2,3,4,5]!
obj.db.test7 = set([1,2,3,4,5])
# in-situ manipulation
obj.db.test8 = [1,2,{"test":1}]
obj.db.test8[0] = 4
obj.db.test8[2]["test"] = 5
# test8 is now [4,2,{"test":5}]
```

Retrieving Mutable objects

A side effect of the way Evennia stores Attributes is that *mutable* iterables (iterables that can be modified in-place after they were created, which is everything except tuples) are handled by custom objects called `_SaverList`, `_SaverDict` etc. These `_Saver...` classes behave just like the normal variant except that they are aware of the database and saves to it whenever new data gets assigned to them. This is what allows you to do things like `self.db.mylist[7] = val` and be sure that the new version of list is saved. Without this you would have to load the list into a temporary variable, change it and then re-assign it to the Attribute in order for it to save.

There is however an important thing to remember. If you retrieve your mutable iterable into another variable, e.g. `mylist2 = obj.db.mylist`, your new variable (`mylist2`) will *still* be a `_SaverList`. This means it will continue to save itself to the database whenever it is updated! While this can be convenient in some cases, it is important to keep in mind so you are not confused by the results.

```
obj.db.mylist = [1,2,3,4]
mylist = obj.db.mylist
mylist[3] = 5 # this will also update database
print mylist # this is now [1,2,3,5]
print obj.db.mylist # this is also [1,2,3,5]
```

To "disconnect" your extracted mutable variable from the database you simply need to convert the `_Saver...` iterable to a normal Python structure. So to convert a `_SaverList`, you use the `list()` function, for a `_SaverDict` you use `dict()` and so on. In the case of "nested" structure, you only have to convert the "outermost" `_Saver..` entity in order to cut the *entire* structure's connection to the database.

```
obj.db.mylist = [1,2,3,4]
mylist = list(obj.db.mylist) # convert to normal list
mylist[3] = 5
print mylist # this is now [1,2,3,5]
print obj.db.mylist # this is still [1,2,3,4]
```

Remember, this is only valid for *mutable* iterables. Immutable objects (strings, numbers, tuples etc) are already disconnected from the database from the onset. So making the outermost iterable into a *tuple* is another way to stop any changes to the structure from updating the database.

```
obj.db.mytup = (1,2,[3,4])
obj.db.mytup[0] = 5 # this fails since tuples are immutable
# this works but will NOT update database since outermost is a tuple
obj.db.mytup[2][1] = 5
print obj.db.mytup[2][1] # this still returns 4, not 5
mytup1 = obj.db.mytup
# mytup1 is already disconnected from database since outermost
# iterable is a tuple, so we can edit the internal list as we want
# without affecting the database.
```

Locking and checking Attributes

Attributes are normally not locked down by default, but you can easily change that for individual Attributes (like those that may be game-sensitive in games with user-level building).

First you need to set a *lock string* on your Attribute. Lock strings are specified Locks. The relevant lock types are

- `attrread` - limits who may read the value of the Attribute
- `attredit` - limits who may set/change this Attribute

You cannot use the `db` handler to modify Attribute object (such as setting a lock on them) - The `db` handler will return the Attribute's *value*, not the Attribute object itself. Instead you use the AttributeHandler and set it to return the object instead of the value:

```
lockstring = "attread:all();attredit:perm(Wizards)"
obj.attributes.get("myattr", return_obj=True).locks.add(lockstring)
```

Note the `return_obj` keyword which makes sure to return the `Attribute` object so its LockHandler could be accessed.

A lock is no good if nothing checks it – and by default Evennia does not check locks on Attributes. You have to add a check to your commands/code wherever it fits (such as before setting an Attribute).

```
# in some command code where we want to limit
# setting of a given attribute name on an object
attr = obj.attributes.get(attrname,
                          return_obj=True,
                          accessing_obj=caller,
                          default=None,
                          default_access=False)
if not attr:
    caller.msg("You cannot edit that Attribute!")
    return
# edit the Attribute here
```

The same keywords are available to use with `obj.attributes.set()` and `obj.attributes.remove()`, those will check for the `attredit` lock type.

Locks

For most games it is a good idea to restrict what people can do. In Evennia such restrictions are applied and checked by something called *locks*. All Evennia entities (Commands, Objects, Scripts, Players, Help System, messages and channels) are accessed through locks.

A lock can be thought of as an "access rule" restricting a particular use of an Evennia entity. Whenever another entity wants that kind of access the lock will analyze that entity in different ways to determine if access should be granted or not. Evennia implements a "lockdown" philosophy - all entities are inaccessible unless you explicitly define a lock that allows some or full access.

Let's take an example: An object has a lock on itself that restricts how people may "delete" that object. Apart from knowing that it restricts deletion, the lock also knows that only players with the specific ID of, say, `34` are allowed to delete it. So whenever a player tries to run `@delete` on the object, the `@delete` command makes sure to check if this player is really allowed to do so. It calls the lock, which in turn checks if the player's id is `34`. Only then will it allow `@delete` to go on with its job.

Setting and checking a lock

The in-game command for setting locks on objects is `@lock`:

```
> @lock obj = <lockstring>
```

The `<lockstring>` is a string of a certain form that defines the behaviour of the lock. We will go into more detail on how `<lockstring>` should look in the next section.

Code-wise, Evennia handles locks through what is usually called `locks` on all relevant entities. This is a handler that allows you to add, delete and check locks.

```
myobj.locks.add(<lockstring>)
```

One can call `locks.check()` to perform a lock check, but to hide the underlying implementation all objects also have a convenience function called `access`. This should preferably be used. In the example below, `accessing_obj` is the object requesting the 'delete' access whereas `obj` is the object that might get deleted. This is how it would (and do) look from inside the `@delete` command:

```
if not obj.access(accessing_obj, 'delete'):
    accessing_obj.msg("Sorry, you may not delete that.")
    return
```

Defining locks

Defining a lock (i.e. an access restriction) in Evennia is done by adding simple strings of lock definitions to the object's `locks` property using `obj.locks.add()`.

Here are some examples of lock strings (not including the quotes):

```
delete:id(34)    # only allow obj #34 to delete
edit:all()       # let everyone edit
# only those who are not "very_weak" or are Wizards may pick this up
get: not attr(very_weak) or perm(Wizards)
```

Formally, a lockstring has the following syntax:

```
access_type: [NOT] lockfunc1([arg1,..]) [AND|OR] [NOT] lockfunc2([arg1,...]) [...]
```

where `[]` marks optional parts. `AND`, `OR` and `NOT` are not case sensitive and excess spaces are ignored. `lockfunc1`, `lockfunc2` etc are special *lock functions* available to the lock system.

So, a lockstring consists of the type of restriction (the `access_type`), a colon (`:`) and then an expression involving function calls that determine what is needed to pass the lock. Each function returns either `True` or `False`. `AND`, `OR` and `NOT` work as they do normally in Python. If the total result is `True`, the lock is passed.

You can create several lock types one after the other by separating them with a semicolon (`;`) in the lockstring. The string below yields the same result as the previous example:

```
delete:id(34);edit:all();get: not attr(very_weak) or perm(Wizard)
```

Valid access_types

An `access_type`, the first part of a lockstring, defines what kind of capability a lock controls, such as "delete" or "edit". You may in principle name your `access_type` anything as long as it is unique for the particular object. The name of the access types is not case-sensitive.

If you want to make sure the lock is used however, you should pick `access_type` names that you (or the default command set) actually checks for, as in the example of `@delete` above that uses the 'delete' `access_type`.

Below are the access_types checked by the default commandset.

- Commands
- `cmd` - this defines who may call this command at all.
- Objects:
- `control` - who is the "owner" of the object. Can set locks, delete it etc. Defaults to the creator of the object.
- `call` - who may call object-commands on this object.
- `examine` - who may examine this object's properties.
- `delete` - who may delete the object.
- `edit` - who may edit properties and attributes of the object.
- `view` - if the `look` command will display/list this object
- `get`- who may pick up the object and carry it around.
- `puppet` - who may "become" this object and control it as their "character".
- `attrcreate` - who may create new attributes on the object (default True)
- Characters:
- Same as for Objects
- Exits:
- Same as for Objects
- `traverse` - who may pass the exit.
- Players:
- `examine` - who may examine the player's properties.
- `delete` - who may delete the player.

- `edit` - who may edit the player's attributes and properties.
- `msg` - who may send messages to the player.
- `boot` - who may boot the player.
- Attributes: (only checked by `obj.secure_attr`)
- `attrread` - see/access attribute
- `attredit` - change/delete attribute
- Channels:
- `control` - who is administrating the channel. This means the ability to delete the channel, boot listeners etc.
- `send` - who may send to the channel.
- `listen` - who may subscribe and listen to the channel.
- HelpEntry:
- `examine` - who may view this help entry (usually everyone)
- `edit` - who may edit this help entry.

So to take an example, whenever an exit is to be traversed, a lock of the type *traverse* will be checked. Defining a suitable lock type for an exit object would thus involve a lockstring `traverse: <lock functions>`.

Lock functions

You are not allowed to use just any function in your lock definition; you are in fact only allowed to use those functions defined in one of the modules given in `settings.LOCK_FUNC_MODULES`. All functions in any of those modules will automatically be considered a valid lock function. The default ones are found in `evennia/locks/lockfuncs.py` or as properties on `evennia.lockfuncs`.

A lock function must always accept at least two arguments - the *accessing object* (this is the object wanting to get access) and the *accessed object* (this is the object with the lock). Those two are fed automatically as the first two arguments the function when the lock is checked. Any arguments explicitly given in the lock definition will appear as extra arguments.

```
# A simple example lock function. Called with e.g. `id(34)`

def id(accessing_obj, accessed_obj, *args, **kwargs):
    if args:
        wanted_id = args[0]
        return accessing_obj.id == wanted_id
    return False
```

(Using the `*` and `**` syntax causes Python to magically put all extra arguments into a list `args` and all keyword arguments into a dictionary `kwargs` respectively. If you are unfamiliar with how `*args` and `**kwargs` work, see the Python manuals). Some useful default lockfuncs (see `src/locks/lockfuncs.py` for more):

- `true()/all()` - give access to everyone
- `false()/none()/superuser()` - give access to none. Superusers bypass the check entirely and are thus the only ones who will pass this check.
- `perm(perm)` - this tries to match a given `permission` property, on a Player firsthand, on a Character second. See below.
- `perm_above(perm)` - like `perm` but requires a "higher" permission level than the one given.

- `id(num)/dbref(num)` - checks so the access_object has a certain dbref/id.
- `attr(attrname)` - checks if a certain Attribute exists on accessing_object.
- `attr(attrname, value)` - checks so an attribute exists on accessing_object *and* has the given value.
- `attr_gt(attrname, value)` - checks so accessing_object has a value larger (>) than the given value.
- `attr_ge, attr_lt, attr_le, attr_ne` - corresponding for >=, <, <= and `!=`.
- `holds(objid)` - checks so the accessing objects contains an object of given name or dbref.
- `inside()` - checks so the accessing object is inside the accessed object (the inverse of `holds()`).
- `pperm(perm)`, `pid(num)/pdbref(num)` - same as `perm`, `id/dbref` but always looks for permissions and dbrefs of *Players*, not on Characters.
- `serversetting(settingname, value)` - Only returns True if Evennia has a given setting or a setting set to a given value.

Checking simple strings

Sometimes you don't really need to look up a certain lock, you just want to check a lockstring. A common use is inside Commands, in order to check if a user has a certain permission. The lockhandler has a method `check_lockstring(accessing_obj, lockstring, bypass_superuser=False)` that allows this.

```
# inside command definition
if not self.caller.locks.check_lockstring(self.caller, "dummy:perm(Wizards)"):
    self.caller.msg("You must be Wizard or higher to do this!")
    return
```

Note here that the `access_type` can be left to a dummy value since this method does not actually do a Lock lookup.

Default locks

Evennia sets up a few basic locks on all new objects and players (if we didn't, noone would have any access to anything from the start). This is all defined in the root Typeclasses of the respective entity, in the hook method `basetype_setup()` (which you usually don't want to edit unless you want to change how basic stuff like rooms and exits store their internal variables). This is called once, before `at_object_creation`, so just put them in the latter method on your child object to change the default. Also creation commands like `@create` changes the locks of objects you create - for example it sets the `control` lock_type so as to allow you, its creator, to control and delete the object.

Permissions

A *permission* is simply a list of text strings stored in the handler `permissions` on `Objects` and `Players`. Permissions can be used as a convenient way to structure access levels and hierarchies. It is set by the `@perm` command. Permissions are especially handled by the `perm()` and `pperm()` lock functions listed above.

Let's say we have a `red_key` object. We also have red chests that we want to unlock with this key.

```
@perm red_key = unlocks_red_chests
```

This gives the `red_key` object the permission "unlocks_red_chests". Next we lock our red chests:

```
@lock red chest = unlock:perm(unlocks_red_chests)
```

What this lock will expect is to the fed the actual key object. The `perm()` lock function will check the permissions set on the key and only return true if the permission is the one given.

Finally we need to actually check this lock somehow. Let's say the chest has an command `open <key>` sitting on itself. Somewhere in its code the command needs to figure out which key you are using and test if this key has the correct permission:

```
# self.obj is the chest
# and used_key is the key we used as argument to
# the command. The self.caller is the one trying
# to unlock the chest
if not self.obj.access(used_key, "unlock"):
    self.caller.msg("The key does not fit!")
    return
```

All new players are given a default set of permissions defined by `settings.PERMISSION_PLAYER_DEFAULT`.

Selected permission strings can be organized in a *permission hierarchy* by editing the tuple `settings.PERMISSION_HIERARCHY`. Evennia's default permission hierarchy is as follows:

```
Immortals        # like superuser but affected by locks
Wizards          # can administrate players
Builders         # can edit the world
PlayerHelpers    # can edit help files
Players          # can chat and send tells (default level)
```

The main use of this is that if you use the lock function `perm()` mentioned above, a lock check for a particular permission in the hierarchy will *also* grant access to those with *higher* hierarchy access. So if you have the permission "Wizards" you will also pass a lock defined as `perm(Builders)` or any of those levels below "Wizards".

When doing an access check from an Object or Character, the `perm()` lock function will always first use the permissions of any Player connected to that Object before checking for permissions on the Object. In the case of hierarchical permissions (Wizards, Builders etc), the Player permission will always be used (this stops a Player from escalating their permission by puppeting a high-level Character). If the permission looked for is not in the hierarchy, an exact match is required, first on the Player and if not found there (or if no Player is connected), then on the Object itself.

Here is how you use `@perm` to give a player more permissions:

```
@perm/player Tommy = Builders
@perm/player/del Tommy = Builders # remove it again
```

Note the use of the `/player` switch. It means you assign the permission to the Players Tommy instead of any Character that also happens to be named "Tommy".

Putting permissions on the *Player* guarantees that they are kept, *regardless* of which Character they are currently puppeting. This is especially important to remember when assigning permissions from the *hierarchy tree* - as mentioned above, a Player's permissions will overrule that of its character. So to be sure to avoid confusion you should generally put hierarchy permissions on the Player, not on their Characters (but see also quelling).

Below is an example of an object without any connected player

```
obj1.permissions = ["Builders", "cool_guy"]
obj2.locks.add("enter:perm_above(Players) and perm(cool_guy)")

obj2.access(obj1, "enter") # this returns True!
```

And one example of a puppet with a connected player:

```
player.permissions.add("Players")
puppet.permissions.add("Builders", "cool_guy")
obj2.locks.add("enter:perm_above(Players) and perm(cool_guy)")

obj2.access(puppet, "enter") # this returns False!
```

Superusers

There is normally only one *superuser* account and that is the one first created when starting Evennia (User #1). This is sometimes known as the "Owner" or "God" user. A superuser has more than full access - it completely *bypasses* all locks so no checks are even run. This allows for the superuser to always have access to everything in an emergency. But it also hides any eventual errors you might have made in your lock definitions. So when trying out game systems you should either use quelling (see below) or make a second Immortal-level character so your locks get tested correctly.

Quelling

The `@quell` command can be used to enforce the `perm()` lockfunc to ignore permissions on the Player and instead use the permissions on the Character only. This can be used e.g. by staff to test out things with a lower permission level. Return to the normal operation with `@unquell`. Note that quelling will use the smallest of any hierarchical permission on the Player or Character, so one cannot escalate one's Player permission by quelling to a high-permission Character. Also the superuser can quell their powers this way, making them affectable by locks.

More Lock definition examples

```
examine: attr(eyesight, excellent) or perm(Builders)
```

You are only allowed to do *examine* on this object if you have 'excellent' eyesight (that is, has an Attribute `eyesight` with the value `excellent` defined on yourself) or if you have the "Builders" permission string assigned to you.

```
open: holds('the green key') or perm(Builder)
```

This could be called by the `open` command on a "door" object. The check is passed if you are a Builder or has the right key in your inventory.

```
cmd: perm(Builders)
```

Evennia's command handler looks for a lock of type `cmd` to determine if a user is allowed to even call upon a particular command or not. When you define a command, this is the kind of lock you must set. See the default command set for lots of examples. If a character/player don't pass the `cmd` lock type the command will not even appear in their `help` list.

```
cmd: not perm(no_tell)
```

"Permissions" can also be used to block users or implement highly specific bans. The above example would be be added as a lock string to the `tell` command. This will allow everyone *not* having the "permission" `no_tell` to use the `tell` command. You could easily give a player the "permission" `no_tell` to disable their use of this particular command henceforth.

```
dbref = caller.id
lockstring = "control:id(%s);examine:perm(Builders);delete:id(%s) or perm(Wizards);
↪get:all()" % (dbref, dbref)
new_obj.locks.add(lockstring)
```

This is how the `@create` command sets up new objects. In sequence, this permission string sets the owner of this object be the creator (the one running `@create`). Builders may examine the object whereas only Wizards and the creator may delete it. Everyone can pick it up.

A complete example of setting locks on an object

Assume we have two objects - one is ourselves (not superuser) and the other is an Object called `box`.

```
> @create/drop box
> @desc box = "This is a very big and heavy box."
```

We want to limit which objects can pick up this heavy box. Let's say that to do that we require the would-be lifter to to have an attribute *strength* on themselves, with a value greater than 50. We assign it to ourselves to begin with.

```
> @set self/strength = 45
```

Ok, so for testing we made ourselves strong, but not strong enough. Now we need to look at what happens when someone tries to pick up the the box - they use the `get` command (in the default set). This is defined in `evennia/commands/default/general.py`. In its code we find this snippet:

```
if not obj.access(caller, 'get'):
    if obj.db.get_err_msg:
        caller.msg(obj.db.get_err_msg)
    else:
        caller.msg("You can't get that.")
    return
```

So the `get` command looks for a lock with the type *get* (not so surprising). It also looks for an Attribute on the checked object called *get_err_msg* in order to return a customized error message. Sounds good! Let's start by setting that on the box:

```
> @set box/get_err_msg = You are not strong enough to lift this box.
```

Next we need to craft a Lock of type *get* on our box. We want it to only be passed if the accessing object has the attribute *strength* of the right value. For this we would need to create a lock function that checks if attributes have a value greater than a given value. Luckily there is already such a one included in evennia (see `evennia/locks/lockfuncs.py`), called `attr_gt`.

So the lock string will look like this: `get:attr_gt(strength, 50)`. We put this on the box now:

```
@lock box = get:attr_gt(strength, 50)
```

Try to `get` the object and you should get the message that we are not strong enough. Increase your strength above 50 however and you'll pick it up no problem. Done! A very heavy box!

If you wanted to set this up in python code, it would look something like this:

```
from evennia import create_object

   # create, then set the lock
   box = create_object(None, key="box")
   box.locks.add("get:attr_gt(strength, 50)")

   # or we can assign locks in one go right away
   box = create_object(None, key="box", locks="get:attr_gt(strength, 50)")

   # set the attributes
```

```
    box.db.desc = "This is a very big and heavy box."
    box.db.get_err_msg = "You are not strong enough to lift this box."

    # one heavy box, ready to withstand all but the strongest...
```

On Django's permission system

Django also implements a comprehensive permission/security system of its own. The reason we don't use that is because it is app-centric (app in the Django sense). Its permission strings are of the form `appname.permstring` and it automatically adds three of them for each database model in the app - for the app evennia/object this would be for example 'object.create', 'object.admin' and 'object.edit'. This makes a lot of sense for a web application, not so much for a MUD, especially when we try to hide away as much of the underlying architecture as possible.

The django permissions are not completely gone however. We use it for validating passwords during login. It is also used exclusively for managing Evennia's web-based admin site, which is a graphical front-end for the database of Evennia. You edit and assign such permissions directly from the web interface. It's stand-alone from the permissions described above.

Help System

An important part of Evennia is the online help system. This allows the players and staff alike to learn how to use the game's commands as well as other information pertinent to the game. The help system has many different aspects, from the normal editing of help entries from inside the game, to auto-generated help entries during code development using the *auto-help system*.

Viewing the help database

The main command is `help`:

```
help [searchstring]
```

This will show a list of help entries, ordered after categories. You will find two sections, *Command help entries* and *Other help entries* (initially you will only have the first one). You can use help to get more info about an entry; you can also give partial matches to get suggestions. If you give category names you will only be shown the topics in that category.

Command Auto-help system

A common item that requires help entries are in-game commands. Keeping these entries up-to-date with the actual source code functionality can be a chore. Evennia's commands are therefore auto-documenting straight from the sources through its *auto-help system*. Only commands that you and your character can actually currently use are picked up by the auto-help system. That means an admin will see a considerably larger amount of help topics than a normal player when using the default `help` command.

The auto-help system uses the `__doc__` strings of your command classes and formats this to a nice-looking help entry. This makes for a very easy way to keep the help updated - just document your commands well and updating the help file is just a `@reload` away. There is no need to manually create and maintain help database entries for commands; as long as you keep the docstrings updated your help will be dynamically updated for you as well.

Example (from a module with command definitions):

```
class CmdMyCmd(Command):
    """
    mycmd - my very own command

    Usage:
      mycmd[/switches] <args>

    Switches:
      test - test the command
      run  - do something else

    This is my own command that does this and that.

    """
    # [...]

    help_category = "General"     # default
    auto_help = True              # default

    # [...]
```

The text at the very top of the command class definition is the class' `__doc__`-string and will be shown to users looking for help. Try to use a consistent format - all default commands are using the structure shown above.

You should also supply the `help_category` class property if you can; this helps to group help entries together for people to more easily find them. See the `help` command in-game to see the default categories. If you don't specify the category, "General" is assumed.

If you don't want your command to be picked up by the auto-help system at all (like if you want to write its docs manually using the info in the next section or you use a cmdset that has its own help functionality) you can explicitly set `auto_help` class property to `False` in your command definition.

Alternatively, you can keep the advantages of *auto-help* in commands, but control the display of command helps. You can do so by overriding the command's `get_help()` method. By default, this method will return the class docstring. You could modify it to add custom behavior: the text returned by this method will be displayed to the character asking for help in this command.

Database help entries

These are all help entries not involving commands (this is handled automatically by the *Command Auto-help system*). Non-automatic help entries describe how your particular game is played - its rules, world descriptions and so on.

A help entry consists of four parts:

- The *topic*. This is the name of the help entry. This is what players search for when they are looking for help. The topic can contain spaces and also partial matches will be found.
- The *help category*. Examples are *Administration*, *Building*, *Comms* or *General*. This is an overall grouping of similar help topics, used by the engine to give a better overview.
- The *text* - the help text itself, of any length.
- locks - a lock definition. This can be used to limit access to this help entry, maybe because it's staff-only or otherwise meant to be restricted. Help commands check for `access_types` `view` and `edit`. An example of a lock string would be `view:perm(Builders)`.

You can create new help entries in code by using `evennia.create_help_entry()`.

```
from evennia import create_help_entry
entry = create_help_entry("emote",
                "Emoting is important because ...",
                category="Roleplaying", locks="view:all()")
```

From inside the game those with the right permissions can use the `@sethelp` command to add and modify help entries.

```
> @sethelp/add emote = The emote command is ...
```

Using `@sethelp` you can add, delete and append text to existing entries. By default new entries will go in the *General* help category. You can change this using a different form of the `@sethelp` command:

```
> @sethelp/add emote, Roleplaying = Emoting is important because ...
```

If the category *Roleplaying* did not already exist, it is created and will appear in the help index.

You can, finally, define a lock for the help entry by following the category with a lock definition:

```
> @sethelp/add emote, Roleplaying, view:all() = Emoting is ...
```

Nicks

Nicks, short for *Nicknames* is a system allowing an object (usually a Player) to assign custom replacement names for other game entities.

Nicks are not to be confused with *Aliases*. Setting an Alias on a game entity actually changes an inherent attribute on that entity, and everyone in the game will be able to use that alias to address the entity thereafter. A *Nick* on the other hand, is used to map a different way *you alone* can refer to that entity. Nicks are also commonly used to replace your input text which means you can create your own aliases to default commands.

Default Evennia use Nicks in three flavours that determine when Evennia actually tries to do the substitution.

- inputline - replacement is attempted whenever you write anything on the command line. This is the default.
- objects - replacement is only attempted when referring to an object
- players - replacement is only attempted when referring a player

Here's how to use it in the default command set (using the `nick` command):

```
nick ls = look
```

This is a good one for unix/linux users who are accustomed to using the `ls` command in their daily life. It is equivalent to `nick/inputline ls = look`.

```
nick/object mycar2 = The red sports car
```

With this example, substitutions will only be done specifically for commands expecting an object reference, such as

```
look mycar2
```

becomes equivalent to "`look The red sports car`".

```
nick/players tom = Thomas Johnsson
```

This is useful for commands searching for players explicitly:

```
@find *tom
```

One can use nicks to speed up input. Below we add ourselves a quicker way to build red buttons. In the future just writing *rb* will be enough to execute that whole long string.

```
nick rb = @create button:examples.red_button.RedButton
```

Nicks could also be used as the start for building a "recog" system suitable for an RP mud.

```
nick/player Arnold = The mysterious hooded man
```

The nick replacer also supports unix-style *templating*:

```
nick build $1 $2 = @create/drop $1;$2
```

This will catch space separated arguments and store them in the the tags `$1` and `$2`, to be inserted in the replacement string. This example allows you to do `build box crate` and have Evennia see `@create/drop box;crate`. You may use any `$` numbers between 1 and 99, but the markers must match between the nick pattern and the replacement.

> If you want to catch "the rest" of a command argument, make sure to put a `$` tag *with no spaces to the right of it* - it will then receive everything up until the end of the line.

You can also use shell-type wildcards:

- * - matches everything.
- ? - matches a single character.
- [seq] - matches everything in the sequence, e.g. [xyz] will match both x, y and z
- [!seq] - matches everything *not* in the sequence. e.g. [!xyz] will match all but x,y z.

Coding with nicks

Nicks are stored as the `Nick` database model and are referred from the normal Evennia object through the `nicks` property - this is known as the *NickHandler*. The NickHandler offers effective error checking, searches and conversion.

```
# A command/channel nick:
  obj.nicks.add("greetjack", "tell Jack = Hello pal!")

# An object nick:
  obj.nicks.add("rose", "The red flower", nick_type="object")

# An player nick:
  obj.nicks.add("tom", "Tommy Hill", nick_type="player")

# My own custom nick type (handled by my own game code somehow):
  obj.nicks.add("hood", "The hooded man", nick_type="my_identsystem")

# get back the translated nick:
 full_name = obj.nicks.get("rose", nick_type="object")

# delete a previous set nick
  object.nicks.remove("rose", nick_type="object")
```

In a command definition you can reach the nick handler through `self.caller.nicks`. See the `nick` command in `evennia/commands/default/general.py` for more examples.

As a last note, The Evennia channel alias systems are using nicks with the `nick_type="channel"` in order to allow users to create their own custom aliases to channels.

Advanced note

Internally, nicks are Attributes saved with the `db_attrype` set to "nick" (normal Attributes has this set to `None`).

The nick stores the replacement data in the Attribute.db_value field as a tuple with four fields `(regex_nick, template_string, raw_nick, raw_template)`. Here `regex_nick` is the converted regex representation of the `raw_nick` and the `template-string` is a version of the `raw_template` prepared for efficient replacement of any `$`- type markers. The `raw_nick` and `raw_template` are basically the unchanged strings you enter to the `nick` command (with unparsed `$` etc).

If you need to access the tuple for some reason, here's how:

```
tuple = obj.nicks.get("nickname", return_tuple=True)
# or, alternatively
tuple = obj.nicks.get("nickname", return_obj=True).value
```

Tags

A common task of a game designer is to organize and find groups of objects and do operations on them. A classic example is to have a weather script affect all "outside" rooms. Another would be for a player casting a magic spell that affects every location "in the dungeon", but not those "outside". Another would be to quickly find everyone joined with a particular guild or everyone currently dead.

Tags are short text labels that you attach to objects so as to easily be able to retrieve and group them. An Evennia entity can be tagged with any number of Tags. On the database side, Tag entities are *shared* between all objects with that tag. This makes them very efficient but also fundamentally different from Attributes, each of which always belongs to one *single* object.

In Evennia, Tags are technically also used to implement `Aliases` (alternative names for objects) and `Permissions` (simple strings for Locks to check for).

Properties of Tags (and Aliases and Permissions)

Tags are *unique*. This means that there is only ever one Tag object with a given key and category. When Tags are assigned to game entities, these entities are actually sharing the same Tag. This means that Tags are not suitable for storing information about a single object - use an Attribute for this instead. Tags are a lot more limited than Attributes but this also makes them very quick to lookup in the database - this is the whole point.

Tags have the following properties, stored in the database:

- **key** - the name of the Tag. This is the main property to search for when looking up a Tag.
- **category** - this category allows for retrieving only specific subsets of tags used for different purposes. You could have one category of tags for "zones", another for "outdoor locations", for example.
- **data** - this is an optional text field with information about the tag. Remember that Tags are shared between entities, so this field cannot hold any object-specific information. Usually it would be used to hold info about the group of entities the Tag is tagging - possibly used for contextual help like a tool tip. It is not used by default.

There are also two special properties. These should usually not need to be changed or set, it is used internally by Evennia to implement various other uses it makes of the `Tag` object:

- **model** - this holds a *natural-key* description of the model object that this tag deals with, on the form *application.modelclass*, for example `objects.objectdb`. It used by the TagHandler of each entity type for correctly storing the data behind the scenes.
- **tagtype** - this is a "top-level category" of sorts for the inbuilt children of Tags, namely *Aliases* and *Permissions*. The Taghandlers using this special field are especially intended to free up the *category* property for any use you desire.

Using Tags

You can tag any *typeclassed* object, namely Objects, Players, Scripts and Channels. General tags are added by the *Taghandler*. The tag handler is accessed as a property `tags` on the relevant entity:

```
mychair.tags.add("furniture")
myroom.tags.add("dungeon#01")
myscript.tags.add("weather", category="climate")
myplayer.tags.add("guestaccount")

mychair.tags.all()  # returns a list of Tags
mychair.tags.remove("furniture")
mychair.tags.clear()
```

Adding a new tag will either create a new Tag or re-use an already existing one. When using `remove`, the `Tag` is not deleted but are just disconnected from the tagged object. This makes for very quick operations. The `clear` method removes (disconnects) all Tags from the object. You can also use the default `@tag` command:

```
@tag mychair = furniture
```

A main use of Tag is to use it to search for tagged database entities. You can retrieve all objects with a given Tag like this in code:

```
import evennia

# search for objects (most common)
objs = evennia.search_tag("furniture")
dungeon = evennia.search_tag("dungeon#01")

# search for scripts
weather = evennia.search_tag_script("weather")
climates = evennia.search_tag_script(category="climate")

# search for players
players = evennia.search_tag_player("guestaccount")
```

There is also an in-game command that deals with assigning and using (Object-) tags:

```
@tag/search furniture
```

Using Aliases and Permissions

Aliases and Permissions are implemented using normal TagHandlers that simply save Tags with a different `tagtype`. These handlers are named `aliases` and `permissions` on all Objects. They are used in the same way as Tags above:

```
boy.aliases.add("rascal")
boy.permissions.add("Builders")
boy.permissions.remove("Builders")

all_aliases = boy.aliases.all()
```

and so on. Similarly to how `@tag` works in-game, there is also the `@perm` command for assigning permissions and `@alias` command for aliases.

Using Tag categories

Generally, tags are enough on their own for grouping objects. Having no tag `category` is perfectly fine and the normal operation. Simply adding a new Tag for grouping objects is often better than making a new category. So think hard before deciding you really need to categorize your Tags.

That said, tag categories can be useful if you build some game system that uses tags. You can then use tag categories to make sure to separate tags created with this system from any other tags created elsewhere. You can then supply custom search methods that *only* find objects tagged with tags of that category. An example of this is found in the Zone tutorial.

Web Features

Evennia is its own webserver and hosts a default website and browser webclient.

Web site

The Evennia website is a Django application that ties in with the MUD database. Since the website shares this database you could, for example, tell website visitors how many players are logged into the game at the moment, how long the server has been up and any other database information you may want. During development you can access the website by pointing your browser to `http://localhost:8000`.

> You may also want to set `DEBUG = True` in your settings file for debugging the website. You will then see proper tracebacks in the browser rather than just error codes. Note however that this will *leak memory a lot* (it stores everything all the time) and is *not to be used in production*. It's recommended to only use `DEBUG` for active web development and to turn it off otherwise.

A Django (and thus Evennia) website basically consists of three parts, a view an associated template and an `urls.py` file. Think of the view as the Python back-end and the template as the HTML files you are served, optionally filled with data from the back-end. The urls file is a sort of mapping that tells Django that if a specific URL is given in the browser, a particular view should be triggered. You are wise to review the Django documentation for details on how to use these components.

Evennia's default website is located in evennia/web/website. In this folder you'll find the simple default view as well as subfolders `templates` and `static`. Static files are things like images, CSS files and Javascript.

Customizing the Website

You customize your website from your game directory. In the folder `web` you'll find folders `static`, `templates`, `static_overrides` and `templates_overrides`. The first two of those are populated automatically by Django and used to serve the website. You should not edit anything in them - the change will be lost. To customize the

website you'll need to copy the file you want to change from the `web/website/template/` or `web/website/static/` path to the corresponding place under one of_overrides' directories.

Example: To override or modify `evennia/web/website/template/website/index.html` you need to add/modify `mygame/web/template_overrides/website/index.html`.

The detailed description on how to customize the website is best described in tutorial form. See the Web Tutorial for more information.

Overloading Django views

The Python backend for every HTML page is called a Django view. A view can do all sorts of functions, but the main one is to update variables data that the page can display, like how your out-of-the-box website will display statistics about number of users and database objects.

To re-point a given page to a `view.py` of your own, you need to modify `mygame/web/urls.py`. An URL pattern is a regular expression that you need to enter in the address field of your web browser to get to the page in question. If you put your own URL pattern *before* the default ones, your own view will be used instead. The file `urls.py` even marks where you should put your change.

Here's an example:

```
# mygame/web/urls.py

from django.conf.urls import url, include
# default patterns
from evennia.web.urls import urlpatterns

# our own view to use as a replacement
from web.myviews import myview

# custom patterns to add
patterns = [
    # overload the main page view
    url(r'^', myview, name='mycustomview'),
]

urlpatterns = patterns + urlpatterns
```

Django will always look for a list named `urlpatterns` which consists of the results of `url()` calls. It will use the *first* match it finds in this list. Above, we add a new URL redirect from the root of the website. It will now our own function `myview` from a new module `mygame/web/myviews.py`.

> If our game is found on `http://mygame.com`, the regular expression `"^"` means we just entered `mygame.com` in the address bar. If we had wanted to add a view for `http://mygame.com/awesome`, the regular expression would have been `^/awesome`.

Look at evennia/web/website/views.py to see the inputs and outputs you must have to define a view. Easiest may be to copy the default file to `mygame/web` to have something to modify and expand on.

Restart the server and reload the page in the browser - the website will now use your custom view. If there are errors, consider turning on `settings.DEBUG` to see the full tracebacks - in debug mode you will also log all requests in `mygame/server/logs/http_requests.log`.

Web client

Evennia comes with a MUD client accessible from a normal web browser. During development you can try it at `http://localhost:8000/webclient`. The client consists of several parts, all under evennia/web/webclient/:

- `templates/webclient/webclient.html` and `templates/base.html` are the very simplistic django html templates describing the webclient layout. The `base.html` acts as a header that sets up all the basic stuff the client needs, so one only needs to modify the much cleaner `webclient.html`.
- `static/webclient/evennia.js` is the main evennia javascript library. This handles all communication between Evennia and the client over websockets and via AJAX/COMET if the browser can't handle websockets. It will make the `Evennia` object available to the javascript namespace, which offers methods for sending and receiving data to/from the server transparently. This is intended to be used also if swapping out the gui front end.
- `static/webclient/js/evennia.js` is the default GUI of the webclient. This handles the input line, the send button and sends text to the right DOM elements in the html file. It makes use of the `evennia.js` library for all in/out and implements a "telnet-like" interface.
- `static/webclient/css/webclient.css` is the CSS file for the client; it also defines things like how to display ANSI/Xterm256 colors etc.
- The server-side webclient protocols are found in `evennia/server/portal/webclient.py` and `webclient_ajax.py` for the two types of connections. You can't (and should not need to) modify these.

Customizing the web client

Like was the case for the website, you override the webclient from your game directory. You need to add/modify a file in the matching directory location within one of the `_overrides` directories.

Example: To replace/modify `evennia/web/webclient/static/webclient/js/webclient_gui.js` you need copy and modify the file at `mygame/web/static_overrides/webclient/js/webclient_gui.js`.

See the Web Tutorial for more on how to customize the webclient.

The Django 'Admin' Page

Django comes with a built-in admin website. This is accessible by clicking the 'admin' button from your game website. The admin site allows you to see, edit and create objects in your database from a graphical interface.

The behavior of default Evennia models are controlled by files `admin.py` in the Evennia package. New database models you choose to add yourself (such as in the Web Character View Tutorial) can/will also have `admin.py` files. New models are registered to the admin website by a call of `admin.site.register(model class, admin class)` inside an admin.py file. It is an error to attempt to register a model that has already been registered.

To overload Evennia's admin files you don't need to modify Evennia itself. To customize you can call `admin.site.unregister(model class)`, then follow that with `admin.site.register` in one of your own admin.py files in a new app that you add.

More reading

Evennia relies on Django for its web features. For details on expanding your web experience, the Django documentation or the Django Book are the main resources to look into. In Django lingo, the Evennia is a django "project" that consists of Django "applications". For the sake of web implementation, the relevant django "applications" in default Evennia are `web/website` or `web/webclient`.

TickerHandler

One way to implement a dynamic MUD is by using "tickers", also known as "heartbeats". A ticker is a timer that fires ("ticks") at a given interval. The tick triggers updates in various game systems.

About Tickers

Tickers are very common or even unavoidable in other mud code bases. Certain code bases are even hard-coded to rely on the concept of the global 'tick'. Evennia has no such notion - the decision to use tickers is very much up to the need of your game and which requirements you have. The "ticker recipe" is just one way of cranking the wheels.

The most fine-grained way to manage the flow of time is of course to use Scripts. Many types of operations (weather being the classic example) are however done on multiple objects in the same way at regular intervals, and for this, storing separate Scripts on each object is inefficient. The way to do this is to use a ticker with a "subscription model" - let objects sign up to be triggered at the same interval, unsubscribing when the updating is no longer desired.

Evennia offers an optimized implementation of the subscription model - the *TickerHandler*. This is a singleton global handler reachable from `evennia.TICKER_HANDLER`. You can assign any *callable* (a function or, more commonly, a method on a database object) to this handler. The TickerHandler will then call this callable at an interval you specify, and with the arguments you supply when adding it. This continues until the callable un-subscribes from the ticker. The handler survives a reboot and is highly optimized in resource usage.

Here is an example of importing `TICKER_HANDLER` and using it:

```
# we assume that obj has a hook "at_tick" defined on itself
from evennia import TICKER_HANDLER as tickerhandler

tickerhandler.add(20, obj.at_tick)
```

That's it - from now on, `obj.at_tick()` will be called every 20 seconds.

You can also import function and tick that:

```
from evennia import TICKER_HANDLER as tickerhandler
from mymodule import myfunc

tickerhandler.add(30, myfunc)
```

Removing (stopping) the ticker works as expected:

```
tickerhandler.remove(20, obj.at_tick)
tickerhandler.remove(30, myfunc)
```

Note that you have to also supply `interval` to identify which subscription to remove. This is because the TickerHandler maintains a pool of tickers and a given callable can subscribe to be ticked at any number of different intervals.

The full definition of the `tickerhandler.add` method is

```
tickerhandler.add(interval, callback,
                  idstring="", persistent=True, *args, **kwargs)
```

Here `*args` and `**kwargs` will be passed to `callback` every `interval` seconds. If `persistent` is `False`, this subscription will not survive a server reload.

Tickers are identified and stored by making a key of the callable itself, the ticker-interval, the `persistent` flag and the `idstring` (the latter being an empty string when not given explicitly).

Since the arguments are not included in the ticker's identification, the `idstring` must be used to have a specific callback triggered multiple times on the same interval but with different arguments:

```
    tickerhandler.add(10, obj.update, "ticker1", True, 1, 2, 3)
    tickerhandler.add(10, obj.update, "ticker2", True, 4, 5)

Note that, when we want to send arguments to our callback within a
ticker handler, we need to specify ``idstring`` and ``persistent``
before, unless we call our arguments as keywords, which would often
be more readable:
```

```
tickerhandler.add(10, obj.update, caller=self, value=118)
```

If you add a ticker with exactly the same combination of callback, interval and idstring, it will overload the existing ticker. This identification is also crucial for later removing (stopping) the subscription:

```
tickerhandler.remove(10, obj.update, idstring="ticker1")
tickerhandler.remove(10, obj.update, idstring="ticker2")
```

The `callable` can be on any form as long as it accepts the arguments you give to send to it in `TickerHandler.add`.

> Note that everything you supply to the TickerHandler will need to be pickled at some point to be saved into the database. Most of the time the handler will correctly store things like database objects, but the same restrictions as for Attributes apply to what the TickerHandler may store.

When testing, you can stop all tickers in the entire game with `tickerhandler.clear()`. You can also view the currently subscribed objects with `tickerhandler.all()`.

See the Weather Tutorial for an example of using the TickerHandler.

When *not* to use TickerHandler

Using the TickerHandler may sound very useful but it is important to consider when not to use it. Even if you are used to habitually relying on tickers for everything in other code bases, stop and think about what you really need it for. This is the main point:

> You should *never* use a ticker to catch *changes*.

Think about it - you might have to run the ticker every second to react to the change fast enough. Most likely nothing will have changed at a given moment. So you are doing pointless calls (since skipping the call gives the same result as doing it). Making sure nothing's changed might even be computationally expensive depending on the complexity of your system. Not to mention that you might need to run the check *on every object in the database*. Every second. Just to maintain status quo ...

Rather than checking over and over on the off-chance that something changed, consider a more proactive approach. Could you implement your rarely changing system to *itself* report when its status changes? It's almost always much cheaper/efficient if you can do things "on demand". Evennia itself uses hook methods for this very reason.

So, if you consider a ticker that will fire very often but which you expect to have no effect 99% of the time, consider handling things things some other way. A self-reporting on-demand solution is usually cheaper also for fast-updating properties. Also remember that some things may not need to be updated until someone actually is examining or using them - any interim changes happening up to that moment are pointless waste of computing time.

The main reason for needing a ticker is when you want things to happen to multiple objects at the same time without input from something else.

Spawner

The *spawner* is a system for defining and creating individual objects from a base template called a *prototype*. It is only designed for use with in-game Objects, not any other type of entity.

The normal way to create a custom object in Evennia is to make a Typeclass. If you haven't read up on Typeclasses yet, think of them as normal Python classes that save to the database behind the scenes. Say you wanted to create a "Goblin" enemy. A common way to do this would be to first create a `Mobile` typeclass that holds everything common to mobiles in the game, like generic AI, combat code and various movement methods. A `Goblin` subclass is then made to inherit from `Mobile`. The `Goblin` class adds stuff unique to goblins, like group-based AI (because goblins are smarter in a group), the ability to panic, dig for gold etc.

But now it's time to actually start to create some goblins and put them in the world. What if we wanted those goblins to not all look the same? Maybe we want grey-skinned and green-skinned goblins or some goblins that can cast spells or which wield different weapons? We *could* make subclasses of `Goblin`, like `GreySkinnedGoblin` and `GoblinWieldingClub`. But that seems a bit excessive (and a lot of Python code for every little thing). Using classes can also become impractical when wanting to combine them - what if we want a grey-skinned goblin shaman wielding a spear - setting up a web of classes inheriting each other with multiple inheritance can be tricky.

This is what the *spawner* is meant for - it offers a simple way to customize *instances* of a typeclass to make them unique and different without needing to modify its typeclass.

Using @spawn

The spawner can be used from inside the game through the Builder-only `@spawn` command:

```
@spawn {"key":"Orc shaman", \
        "typeclass": "typeclasses.monsters.Orc", \
        "weapon": "wooden staff", \
        "health": 20}
```

(The command is all on one line, it's split here for readability). The dictionary should be a valid Python dictionary, including quotes for strings etc. We refer to this dictionary as the *prototype*. Each key of the prototype is parsed and used to customize the creation of the object, much like a blueprint.

One can also put the prototype dictionary in a module. This must be a module included in the `settings.PROTOTYPE_MODULES` list. The default place to put prototypes is in `mygame/world/prototypes.py`. Here is an example of the Orc shaman prototype stored in the module:

```
# in a module Evennia looks at for prototypes,
# (like mygame/server/conf/prototypes.py)

ORC_SHAMAN = {"key":"Orc shaman",
              "typeclass": "typeclasses.monsters.Orc",
              "weapon": "wooden staff",
              "health": 20}
```

With this prototype in place (and assuming you actually have the `Orc` typeclass in `mygame/typeclasses/monsters.py`), you can henceforth build an orc shaman in-game with

```
@spawn ORC_SHAMAN
```

or the equivalent but longer

```
@spawn {"prototype":ORC_SHAMAN}
```

This second form actually creates a new prototype that starts out identical to its parent. This form is useful since prototypes supports inheritance; One can replace any of the keys of the parent prototype on the fly:

```
@spawn {"prototype":"ORC_SHAMAN", \
        "key":"Orguth the terrible", \
        "health":40}
```

(again, the command is one line, we split it for readability in the wiki).

Defining the prototype dictionary

The prototype dictionary allows the following (optional) keys and corresponding values:

> Note: All keywords below can also be asssigned a *callable* rather than a fixed value. If so, that callable will be called without arguments for *every time* a new object of this prototype is created. This is for example ideal for assigning random values.

- `key` - the main object identifier. Defaults to "Spawned Object *X*", where *X* is a random integer.
- `typeclass` - python-path to the typeclass to use, if not set, will use `settings.BASE_OBJECT_TYPECLASS`.
- `location` - this should be a `#dbref`. If not specified, the object will be created in the same location as the one calling `@spawn`. Can be None if created OOC. Note that if you use the spawner functionality in code (see below), no `location` will be set by default.
- `home` - a valid `#dbref`. Defaults to `location` or `settings.DEFAULT_HOME` if location does not exist.
- `destination` - a valid `#dbref`. Only used by exits.
- `permissions` - string or list of permission strings, like `["Players", "may_use_red_door"]`
- `locks` - a lock-string.
- `aliases` - string or list of strings for use as aliases
- `tags` - string or list of strings for use as Tags. These will have no set tag category.
- `prototype` - points to the name of a parent prototype dictionary. The parent slots are used if not overloaded in this prototype. Inheritance will recursively follow the tree until it comes to a prototype without the "prototype" property. Be careful to avoid inheritance loops! The prototype can also be a list for multiple inheritance. Multiple inheritance goes from left to right, with rightmost overloading leftmost parent. The parent prototypes are pre-defined as global variables in a module set by `settings.PROTOTYPE_MODULES` (see below).
- `ndb_<name>` - sets the value of a non-persistent attribute ("ndb_" is stripped from the name).
- `exec` - This accepts a code snippet or a list of code snippets. They will all be called *after* the object has been created in the database and are intended to be used e.g. to call custom methods or handlers on the object. The execution environment has access to `evennia` for the main library and `obj` to access the just created object. Note that `exec` is only available to users of `@spawn` with the permission *Immortals* - the execution of arbitrary python code would be a severe security issue to allow for regular users.
- any other keywords are interpreted as Attributes and their values.

Prototypes from modules

You can add new prototypes by adding a new prototype to the `world/prototypes.py` file in your game directory. You can also add more prototype modules by appending it to the list `PROTOTYPE_MODULES` in your settings file.

All *globally defined dictionaries* in these modules (that is, all dictionaries assigned to variables in the outermost scope of the module) will be read in and interpreted as prototypes. The spawner will import them into a global dictionary where the variable names are keys, and so they can be accessed by their variable name using the `prototype` slot.

In a module the prototype dictionary can be more advanced since you are not limited to input on the command line. Instead of just giving `#dbrefs` you could import and assign objects directly if you wanted to. Each value can also be a callable to allow for dynamic allocation and look-up. The callable takes no argument and should return the value to use for the slot. It will be called every time the prototype
is used.

Below is an example of a prototype module (we assume a `Troll` typeclass already exists in `typeclasses.monsters`):

```
# a file my_prototypes.py

from random import randint

# a base troll creature
TROLL = {"key": "Troll warrior",
         "typeclass": "typeclasses.monsters.Troll",
         "health": lambda:randint(20, 40) # random value between 20-40
         "attacks": ["fists"]
         "resists": ["poison", "cold"]
         "weaknesses": ["fire", "light"],
         "regeneration_speed": 1}

# A troll with a different weapon
TROLL_SMASHER = {"prototype": "TROLL",
                 "key": "Troll smasher",
                 "attacks": ["giant club"]}

# Troll magic user
TROLL_SHAMAN = {"prototype": "TROLL",
                "key": "Troll shaman",
                "spells": ["ice cone", "poison dart"]}
# Boss creature
ELITE_TROLL = {"health": randint(40, 50),
               "attacks": ["fists", "charge"],
               "regeneration_speed": 2,
               "exec": "obj.growl_menacingly()"}
# Magical boss
SHAMAN_BOSS = {"prototype": ("TROLL_SHAMAN", "ELITE_TROLL"),
               "key": "Master shaman"}

# Shaman with a club
TROLL_SHAMAN_SMASHER = {"prototype": ("TROLL_SMASHER", "SHAMAN_BOSS"),
                        "key": "Grunt the bone crusher"}
```

See `mygame/world/prototypes.py` for an example involving goblins.

> **Note:** how in this example `ELITE_TROLL` has no `prototype` of its own. This still makes `ELITE_TROLL` useful in order to decorate other prototypes (like we do for the shaman boss). But if you wanted to create an `ELITE_TROLL` on its own you would get a normal Object and not a Troll since `prototype` is not set to point back to a prototype where the `Troll` typeclass is defined.
>
> **Second Note:** In the above example we define the base "health" of the TROLL to be randomly assigned.

Since this is a callable function, it will be called *every* time a new instance of this prototype (or its children) is created, giving each a random health value.

With the above prototypes loaded you could then use `@spawn` to easily add a troll shaman from the prototype:

```
@spawn TROLL_SHAMAN
```

You could also customize the prototype directly on the command line:

```
@spawn {"prototype":"TROLL_SHAMAN", "key":"Gorat the wise", "tags"=["evil", "quest-
↪giver"]}
```

This would spawn a new troll in the same location. It would get the custom key but otherwise be initialized with all the properties associated with a `TROLL_SHAMAN` (and, by inheritance, that of a `TROLL` as well).

If you wanted you could easily expand the `@spawn` command idea to give your builders custom create-commands based on various prototypes.

Using evennia.utils.spawner()

In code you access the spawner mechanism directly via the call

```
new_objects = evennia.utils.spawner.spawn(*prototype)
```

All arguments are prototype dictionaries. The function will return a
matching list of created objects. Example:

```
obj1, obj2 = evennia.utils.spawner.spawn({"key": "Obj1", "desc": "A test"},
                                         {"key": "Obj2", "desc": "Another test"})
```

Note that no `location` will be set automatically when using `evennia.utils.spawner.spawn()`, you have to specify `location` explicitly in the prototype dict.

If the prototypes you supply are using `parent` keywords, the spawner will look to `settings.PROTOTYPE_MODULES` to determine which modules contain parents available to use. You can use the `prototype_modules` keyword to change the list of available parent modules only for this particular call. Finally, calling `spawn(return_prototypes=True)` will return a dictionary of all the available prototypes from all available modules. In this case, no objects will be created or returned - this is meant to be used for compiling help information for an end user.

CHAPTER 12

Coding utilities

This chapter explains various coding utilities and coding practices useful for working with Evennia.

New Models

Note: This is considered an advanced topic.

Evennia offers many convenient ways to store object data, such as via Attributes or Scripts. This is sufficient for most use cases. But if you aim to build a large stand-alone system, trying to squeeze your storage requirements into those may be more complex than you bargain for. Examples may be to store guild data for guild members to be able to change, tracking the flow of money across a game-wide economic system or implement other custom game systems that requires the storage of custom data in a quickly accessible way. Whereas Tags or Scripts can handle many situations, sometimes things may be easier to handle by adding your own database model.

Overview of database tables

SQL-type databases (which is what Evennia supports) are basically highly optimized systems for retrieving text stored in tables. A table may look like this

```
 id | db_key    | db_typeclass_path          | db_permissions  ...
-------------------------------------------------------------------
 1  |  Griatch  | evennia.DefaultCharacter   | Immortals       ...
 2  |  Rock     | evennia.DefaultObject      | None            ...
```

Each line is considerably longer in your database. Each column is referred to as a "field" and every row is a separate object. You can check this out for yourself. If you use the default sqlite3 database, go to your game folder and run

```
evennia dbshell
```

You will drop into the database shell. While there, try:

```
sqlite> .help       # view help

sqlite> .tables     # view all tables

# show the table field names for objects_objectdb
sqlite> .schema objects_objectdb

# show the first row from the objects_objectdb table
sqlite> select * from objects_objectdb limit 1;

sqlite> .exit
```

Evennia uses Django, which abstracts away the database SQL manipulation and allows you to search and manipulate your database entirely in Python. Each database table is in Django represented by a class commonly called a *model* since it describes the look of the table. In Evennia, Objects, Scripts, Channels etc are examples of Django models that we then extend and build on.

Adding a new database table

Here is how you add your own database table/models:

1. In Django lingo, we will create a new "application" - a subsystem under the main Evennia program. For this example we'll call it "myapp". Run the following (you need to have a working Evennia running before you do this, so make sure you have run the steps in Getting Started first):

   ```
   cd mygame/world
   evennia startapp myapp
   ```

2. A new folder `myapp` is created. "myapp" will also be the name (the "app label") from now on. We chose to put it in the `world/` subfolder here, but you could put it in the root of your `mygame` if that makes more sense.
3. The `myapp` folder contains a few empty default files. What we are interested in for now is `models.py`. In `models.py` you define your model(s). Each model will be a table in the database. See the next section and don't continue until you have added the models you want.
4. You now need to tell Evennia that the models of your app should be a part of your database scheme. Add this line to your `mygame/server/conf/settings.py`file (make sure to use the path where you put `myapp` and don't forget the comma at the end of the tuple):

   ```
   INSTALLED_APPS = INSTALLED_APPS + ("world.myapp", )
   ```

5. From `mygame/`, run

   ```
   evennia makemigrations myapp
   evennia migrate
   ```

This will add your new database table to the database. If you have put your game under version control (and you should), don't forget to `git add myapp/*` to add all items to version control.

Defining your models

A Django *model* is the Python representation of a database table. It can be handled like any other Python class. It defines *fields* on itself, objects of a special type. These become the "columns" of the database table. Finally, you create new instances of the model to add new rows to the database.

We won't describe all aspects of Django models here, for that we refer to the vast Django documentation on the subject. Here is a (very) brief example:

```
from django.db import models

class MyDataStore(models.Model):
    "A simple model for storing some data"
    db_key = models.CharField(max_length=80, db_index=True)
    db_category = models.CharField(max_length=80, null=True, blank=True)
    db_text = models.TextField(null=True, blank=True)
    # we need this one if we want to be
    # able to store this in an Evennia Attribute!
    db_date_created = models.DateTimeField('date created', editable=False,
                                           auto_now_add=True, db_index=True)
```

We create four fields: two character fields of limited length and one text field which has no maximum length. Finally we create a field containing the current time of us creating this object.

> The `db_date_created` field, with exactly this name, is *required* if you want to be able to store instances of your custom model in an Evennia Attribute. It will automatically be set upon creation and can after that not be changed. Having this field will allow you to do e.g. `obj.db.myinstance = mydatastore`. If you know you'll never store your model instances in Attributes the `db_date_created` field is optional.

You don't *have* to start field names with `db_`, this is an Evennia convention. It's nevertheless recommended that you do use `db_`, partly for clarity and consistency with Evennia (if you ever want to share your code) and partly for the case of you later deciding to use Evennia's `SharedMemoryModel` parent down the line.

The field keyword `db_index` creates a *database index* for this field, which allows quicker lookups, so it's recommended to put it on fields you know you'll often use in queries. The `null=True` and `blank=True` keywords means that these fields may be left empty or set to the empty string without the database complaining. There are many other field types and keywords to define them, see django docs for more info.

Similar to using django-admin you are able to do `evennia inspectdb` to get an automated listing of model information for an existing database. As is the case with any model generating tool you should only use this as a starting point for your models.

Creating a new model instance

To create a new row in your table, you instantiate the model and then call its `save()` method:

```
from evennia.myapp import MyDataStore

new_datastore = MyDataStore(db_key="LargeSword",
                            db_category="weapons",
                            db_text="This is a huge weapon!")
# this is required to actually create the row in the database!
new_datastore.save()
```

Note that the `db_date_created` field of the model is not specified. Its flag `at_now_add=True` makes sure to set it to the current date when the object is created (it can also not be changed further after creation).

When you update an existing object with some new field value, remember that you have to save the object afterwards, otherwise the database will not update:

```
my_datastore.db_key = "Larger Sword"
my_datastore.save()
```

Evennia's normal models don't need to explicitly save, since they are based on `SharedMemoryModel` rather than the raw django model. This is covered in the next section.

Using the `SharedMemoryModel` parent

Evennia doesn't base most of its models on the raw `django.db.models` but on the Evennia base model `evennia.utils.idmapper.models.SharedMemoryModel`. There are two main reasons for this:

1. Ease of updating fields without having to explicitly call `save()`
2. On-object memory persistence and database caching

The first (and least important) point means that as long as you named your fields `db_*`, Evennia will automatically create field wrappers for them. This happens in the model's Metaclass so there is no speed penalty for this. The name of the wrapper will be the same name as the field, minus the `db_` prefix. So the `db_key` field will have a wrapper property named `key`. You can then do:

```
my_datastore.key = "Larger Sword"
```

and don't have to explicitly call `save()` afterwards. The saving also happens in a more efficient way under the hood, updating only the field rather than the entire model using django optimizations. Note that if you were to manually add the property or method `key` to your model, this will be used instead of the automatic wrapper and allows you to fully customize access as needed.

To explain the second and more important point, consider the following example using the default Django model parent:

```
shield = MyDataStore.objects.get(db_key="SmallShield")
shield.cracked = True # where cracked is not a database field
```

And then later:

```
shield = MyDataStore.objects.get(db_key="SmallShield")
print shield.cracked  # error!
```

The outcome of that last print statement is *undefined*! It could *maybe* randomly work but most likely you will get an `AttributeError` for not finding the `cracked` property. The reason is that `cracked` doesn't represent an actual field in the database. It was just added at run-time and thus Django don't care about it. When you retrieve your shield-match later there is *no* guarantee you will get back the *same Python instance* of the model where you defined `cracked`, even if you search for the same database object.

Evennia relies heavily on on-model handlers and other dynamically created properties. So rather than using the vanilla Django models, Evennia uses `SharedMemoryModel`, which levies something called *idmapper*. The idmapper caches model instances so that we will always get the *same* instance back after the first lookup of a given object. Using idmapper, the above example would work fine and you could retrieve your `cracked` property at any time until you rebooted.

Using the idmapper is both more intuitive and more efficient *per object*; it leads to a lot less reading from disk. The drawback is that this system tends to be more memory hungry *overall*. So if you know that you'll *never* need to add new properties to running instances, or know that you will create new objects all the time yet rarely access them again (like for a log system), the `SharedMemoryModel` you are probably better off making "plain" models rather than using `SharedMemoryModel` and its idmapper.

To use the idmapper and the field-wrapper functionality you just have to have your model classes inherit from `evennia.utils.idmapper.models.SharedMemoryModel` instead of from the default `django.db.models.Model`:

```
from evennia.utils.idmapper.models import SharedMemoryModel

class MyDataStore(SharedMemoryModel):
    # the rest is the same as before, but db_* is important; these will
    # later be settable as .key, .category, .text ...
    db_key = models.CharField(max_length=80, db_index=True)
    db_category = models.CharField(max_length=80, null=True, blank=True)
    db_text = models.TextField(null=True, blank=True)
    db_date_created = models.DateTimeField('date created', editable=False,
                                           auto_now_add=True, db_index=True)
```

Searching for your models

To search your new custom database table you need to use its database *manager* to build a *query*. Note that even if you use `SharedMemoryModel` as described in the previous section, you have to use the actual *field names* in the query, not the wrapper name (so `db_key` and not just `key`).

```
from world.myapp import MyDataStore

# get all datastore objects exactly matching a given key
matches = MyDataStore.objects.filter(db_key="Larger Sword")
# get all datastore objects with a key containing "sword"
# and having the category "weapons" (both ignoring upper/lower case)
matches2 = MyDataStore.objects.filter(db_key__icontains="sword",
                                      db_category__iequals="weapons")
# show the matching data (e.g. inside a command)
for match in matches2:
   self.caller.msg(match.db_text)
```

See the Django query documentation for a lot more information about querying the database.

Execute Python Code

The `@py` command supplied with the default command set of Evennia allows you to execute Python commands directly from inside the game. An alias to `@py` is simply "`!`". *Access to the ''@py'' command should be severely restricted.* This is no joke - being able to execute arbitrary Python code on the server is not something you should entrust to just anybody.

```
@py 1+2
<<< 3
```

Available variables

A few local variables are made available when running `@py`. These offer entry into the running system.

- **self / me** - the calling object (i.e. you)
- **here** - the current caller's location
- **obj** - a dummy Object instance
- **evennia** - Evennia's flat API - through this you can access all of Evennia.

For accessing other objects in the same room you need to use `self.search(name)`. For objects in other locations, use one of the `evennia.search_*` methods. See below.

Returning output

This is an example where we import and test one of Evennia's utilities found in `src/utils/utils.py`, but also accessible through `ev.utils`:

```
@py from ev import utils; utils.time_format(33333)
<<< Done.
```

Note that we didn't get any return value, all we where told is that the code finished executing without error. This is often the case in more complex pieces of code which has no single obvious return value. To see the output from the `time_format()` function we need to tell the system to echo it to us explicitly with `self.msg()`.

```
@py from ev import utils; self.msg(str(utils.time_format(33333)))
09:15
<<< Done.

Warning: When using the ``msg`` function wrap our argument in
``str()`` to convert it into a string above. This is not strictly
necessary for most types of data (Evennia will usually convert to a
string behind the scenes for you). But for *lists* and *tuples* you
will be confused by the output if you don't wrap them in ``str()``:
only the first item of the iterable will be returned. This is
because doing ``msg(text)`` is actually just a convenience shortcut;
the full argument that ``msg`` accepts is something called an
*outputfunc* on the form ``(cmdname, (args), {kwargs})`` (see `the
message path`_ for more info). Sending a list/tuple confuses Evennia
to think you are sending such a structure. Converting it to a string
however makes it clear it should just be displayed as-is.
```

If you were to use Python's standard `print`, you will see the result in your current `stdout` (your terminal by default, otherwise your log file).

Finding objects

A common use for `@py` is to explore objects in the database, for debugging and performing specific operations that are not covered by a particular command.

Locating an object is best done using `self.search()`:

```
@py self.search("red_ball")
<<< Ball

@py self.search("red_ball").db.color = "red"
<<< Done.

@py self.search("red_ball").db.color
<<< red
```

`self.search()` is by far the most used case, but you can also search other database tables for other Evennia entities like scripts or configuration entities. To do this you can use the generic search entries found in `ev.search_*`.

```
@py evennia.search_script("sys_game_time")
<<< [<src.utils.gametime.GameTime object at 0x852be2c>]
```

(Note that since this becomes a simple statement, we don't have to wrap it in `self.msg()` to get the output). You can also use the database model managers directly (accessible through the `objects` properties of database models or as `evennia.managers.*`). This is a bit more flexible since it gives you access to the full range of database search methods defined in each manager.

```
@py evennia.managers.scripts.script_search("sys_game_time")
<<< [<src.utils.gametime.GameTime object at 0x852be2c>]
```

The managers are useful for all sorts of database studies.

```
@py ev.managers.configvalues.all()
<<< [<ConfigValue: default_home]>, <ConfigValue:site_name>, ...]
```

Testing code outside the game

`@py` has the advantage of operating inside a running server (sharing the same process), where you can test things in real time. Much of this *can* be done from the outside too though.

In a terminal, cd to the top of your game directory (this bit is important since we need access to your config file) and run

```
evennia shell
```

Your default Python interpreter will start up, configured to be able to work with and import all modules of your Evennia installation. From here you can explore the database and test-run individual modules as desired.

It's recommended that you get a more fully featured Python interpreter like iPython. If you use a virtual environment, you can just get it with `pip install ipython`. IPython allows you to better work over several lines, and also has a lot of other editing features, such as tab-completion and `__doc__`-string reading.

```
$ evennia shell

IPython 0.10 -- An enhanced Interactive Python
...

In [1]: import evennia
In [2]: evennia.managers.objects.all()
Out[3]: [<ObjectDB: Harry>, <ObjectDB: Limbo>, ...]
```

See the page about the Evennia-API for more things to explore.

Profiling

This is considered an advanced topic mainly of interest to server developers.

Introduction

Sometimes it can be useful to try to determine just how efficient a particular piece of code is, or to figure out if one could speed up things more than they are. There are many ways to test the performance of Python and the running server.

Before digging into this section, remember Donald Knuth's words of wisdom:

> *[...]about 97% of the time: Premature optimization is the root of all evil.*

That is, don't start to try to optimize your code until you have actually identified a need to do so. This means your code must actually be working before you start to consider optimization. Optimization will also often make your code more complex and harder to read. Consider readability and maintainability and you may find that a small gain in speed is just not worth it.

Simple timer tests

Python's `timeit` module is very good for testing small things. For example, in order to test if it is faster to use a `for` loop or a list comprehension you could use the following code:

```
 import timeit
 # Time to do 1000000 for loops
 timeit.timeit("for i in range(100):\n    a.append(i)", setup="a = []")
<<< 10.70982813835144
 # Time to do 1000000 list comprehensions
 timeit.timeit("a = [i for i in range(100)]")
<<<  5.358283996582031
```

The `setup` keyword is used to set up things that should not be included in the time measurement, like `a = []` in the first call.

By default the `timeit` function will re-run the given test 1000000 times and returns the *total time* to do so (so *not* the average per test). A hint is to not use this default for testing something that includes database writes - for that you may want to use a lower number of repeats (say 100 or 1000) using the `number=100` keyword.

Using cProfile

Python comes with its own profiler, named cProfile (this is for cPython, no tests have been done with `pypy` at this point). Due to the way Evennia's processes are handled, there is no point in using the normal way to start the profiler (`python -m cProfile evennia.py`). Instead you start the profiler through the launcher:

```
evennia --profiler start
```

This will start Evennia with the Server component running (in daemon mode) under cProfile. You could instead try `--profile` with the `portal` argument to profile the Portal (you would then need to start the Server separately).

Please note that while the profiler is running, your process will use a lot more memory than usual. Memory usage is even likely to climb over time. So don't leave it running perpetually but monitor it carefully (for example using the `top` command on Linux or the Task Manager's memory display on Windows).

Once you have run the server for a while, you need to stop it so the profiler can give its report. Do *not* kill the program from your task manager or by sending it a kill signal - this will most likely also mess with the profiler. Instead either use `evennia.py stop` or (which may be even better), use `@shutdown` from inside the game.

Once the server has fully shut down (this may be a lot slower than usual) you will find that profiler has created a new file `mygame/server/logs/server.prof`.

Analyzing the profile

The `server.prof` file is a binary file. There are many ways to analyze and display its contents, all of which has only been tested in Linux (If you are a Windows/Mac user, let us know what works).

We recommend the

Runsnake visualizer to see the processor usage of different processes in a graphical form. For more detailed listing of usage time, you can use KCachegrind. To make KCachegrind work with Python profiles you also need the wrapper script pyprof2calltree. You can get pyprof2calltree via `pip` whereas KCacheGrind is something you need to get via your package manager or their homepage.

How to analyze and interpret profiling data is not a trivial issue and depends on what you are profiling for. Evennia being an asynchronous server can also confuse profiling. Ask on the mailing list if you need help and be ready to be able to supply your `server.prof` file for comparison, along with the exact conditions under which it was obtained.

The Dummyrunner

It is difficult to test "actual" game performance without having players in your game. For this reason Evennia comes with the *Dummyrunner* system. The Dummyrunner is a stress-testing system: a separate program that logs into your game with simulated players (aka "bots" or "dummies"). Once connected these dummies will semi-randomly perform various tasks from a list of possible actions. Use `Ctrl-C` to stop the Dummyrunner.

> Warning: You should not run the Dummyrunner on a production database. It will spawn many objects and also needs to run with general permissions.

To launch the Dummyrunner, first start your server normally (with or without profiling, as above). Then start a new terminal/console window and active your virtualenv there too. In the new terminal, try to connect 10 dummy players:

```
evennia --dummyrunner 10
```

The first time you do this you will most likely get a warning from Dummyrunner. It will tell you to copy an import string to the end of your settings file. Quit the Dummyrunner (`Ctrl-C`) and follow the instructions. Restart Evennia and try `evennia --dummyrunner 10` again. Make sure to remove that extra settings line when running a public server.

The actions perform by the dummies is controlled by a settings file. The default Dummyrunner settings file is `evennia/server/server/profiling/dummyrunner_settings.py` but you shouldn't modify this directly. Rather create/copy the default file to `mygame/server/conf/` and modify it there. To make sure to use your file over the default, add the following line to your settings file:

```
DUMMYRUNNER_SETTINGS_MODULE = "server/conf/dummyrunner_settings.py"

Hint: Don't start with too many dummies. The Dummyrunner defaults to
taxing the server much more intensely than an equal number of human
players. A good dummy number to start with is 10-100.
```

Once you have the dummyrunner running, stop it with `Ctrl-C`.

Generally, the dummyrunner system makes for a decent test of general performance; but it is of course hard to actually mimic human user behavior. For this, actual real-game testing is required.

Unit Testing

Unit testing means testing components of a program in isolation from each other to make sure every part works on its own before using it with others. Extensive testing helps avoid new updates causing unexpected side effects as well as alleviates general code rot (a more comprehensive wikipedia article on unit testing can be found here).

A typical unit test sets calls some function or method with a given input, looks at the result and makes sure that this result looks as expected. Rather than having lots of stand-alone test programs, Evennia makes use of a central *test*

runner. This is a program that gathers all available tests all over the Evennia source code (called *test suites*) and runs them all in one go. Errors and tracebacks are reported.

By default Evennia only tests itself. But you can also add your own tests to your game code and have Evennia run those for you.

Running the Evennia test suite

To run the full Evennia test suite, go to your game folder and issue the command

```
evennia test evennia
```

This will run all the evennia tests using the default settings. You could also run only a subset of all tests by specifying a subpackage of the library:

```
evennia test evennia.commands.default
```

A temporary database will be instantiated to manage the tests. If everything works out you will see how many tests were run and how long it took. If something went wrong you will get error messages. If you contribute to Evennia, this is a useful sanity check to see you haven't introduced an unexpected bug.

Running tests with custom settings file

If you have implemented your own tests for your game (see below) you can run them from your game dir with

```
evennia test .
```

The period (`.`) means to run all tests found in the current directory and all subdirectories. You could also specify, say, `typeclasses` or `world` if you wanted to just run tests in those subdirs.

Those tests will all be run using the default settings. To run the tests with your own settings file you must use the `--settings` option:

```
evennia --settings settings.py test .
```

The `--settings` option of Evennia takes a file name in the `mygame/server/conf` folder. It is normally used to swap settings files for testing and development. In combination with `test`, it forces Evennia to use this settings file over the default one.

Writing new tests

Evennia's test suite makes use of Django unit test system, which in turn relies on Python's *unittest* module.

> If you want to help out writing unittests for Evennia, take a look at Evennia's coveralls.io page. There you see which modules have any form of test coverage and which does not.

To make the test runner find the tests, they must be put in a module named `test*.py` (so `test.py`, `tests.py` etc). Such a test module will be found wherever it is in the package. It can be a good idea to look at some of Evennia's `tests.py` modules to see how they look.

Inside a testing file, a `unittest.TestCase` class is used to test a single aspect or component in various ways. Each test case contains one ore more *test methods* - these define the actual tests to run. You can name the test methods anything you want as long as the name starts with "`test_`". Your `TestCase` class can also have a method `setUp()`. This is run before each test, setting up and storing whatever preparations the test methods need. Conversely, a `tearDown()` method can optionally do cleanup after each test.

To test the results, you use special methods of the `TestCase` class. Many of those start with "`assert`", such as `assertEqual` or `assertTrue`.

Example of a `TestCase` class:

```
import unittest

# the function we want to test
from mypath import myfunc

class TestObj(unittest.TestCase):
   "This tests a function myfunc."

   def test_return_value(self):
       "test method. Makes sure return value is as expected."
       expected_return = "This is me being nice."
       actual_return = myfunc()
       # test
       self.assertEqual(expected_return, actual_return)
   def test_alternative_call(self):
       "test method. Calls with a keyword argument."
       expected_return = "This is me being baaaad."
       actual_return = myfunc(bad=True)
       # test
       self.assertEqual(expected_return, actual_return)
```

You might also want to read the documentation for the unittest module.

Using the EvenniaTest class

Evennia offers a custom TestCase, the `evennia.utils.test_resources.EvenniaTest` class. This class initiates a range of useful properties on themselves for testing Evennia systems. Examples are `.player` and `.session` representing a mock connected Player and its Session and `.char1` and `char2` representing Characters complete with a location in the test database. These are all useful when testing Evennia system requiring any of the default Evennia typeclasses as inputs. See the full definition of the `EvenniaTest` class in evennia/utils/test_resources.py.

```
# in a test module

from evennia.utils.test_resources import EvenniaTest

class TestObject(EvenniaTest):
    def test_object_search(self):
        # char1 and char2 are both created in room1
        self.assertEqual(self.char1.search(self.char2.key), self.char2)
        self.assertEqual(self.char1.search(self.char1.location.key), char1.location)
        # ...
```

Testing in-game Commands

In-game Commands are a special case. Tests for the default commands are put in `evennia/commands/default/tests.py`. This uses a custom `CommandTest` class that inherits from `evennia.utils.test_resources.EvenniaTest` described above. `CommandTest` supplies extra convenience functions for executing commands and check that their return values (calls of `msg()` returns expected values. It uses Characters and Sessions generated on the `EvenniaTest` class to call each class).

Each command tested should have its own `TestCase` class. Inherit this class from the `CommandTest` class in the same module to get access to the command-specific utilities mentioned.

```
from evennia.commands.default.tests import CommandTest
from evennia.commands.default import general
class TestSet(CommandTest):
    "tests the look command by simple call"
    def test_mycmd(self):
        self.call(general.CmdLook(), "here", "Room\nroom_desc")
```

Unit testing contribs with custom models

A special case is if you were to create a contribution to go to the `evennia/contrib` folder that uses its own database models. The problem with this is that Evennia (and Django) will only recognize models in `settings.INSTALLED_APPS`. If a user wants to use your contrib, they will be required to add your models to their settings file. But since contribs are optional you cannot add the model to Evennia's central `settings_default.py` file - this would always create your optional models regardless of if the user wants them. But at the same time a contribution is a part of the Evennia distribution and its unit tests should be run with all other Evennia tests using `evennia test evennia`.

The way to do this is to only temporarily add your models to the `INSTALLED_APPS` directory when the test runs. here is an example of how to do it.

> Note that this solution, derived from this stackexchange answer is currently untested! Please report your findings.

```
# a file contrib/mycontrib/tests.py

from django.conf import settings
import django
from evennia.utils.test_resources import EvenniaTest

OLD_DEFAULT_SETTINGS = settings.INSTALLED_APPS
DEFAULT_SETTINGS = dict(
    INSTALLED_APPS=(
        'contrib.mycontrib.tests',
        ),
    DATABASES={
        "default": {
            "ENGINE": "django.db.backends.sqlite3"
            }
        },
    SILENCED_SYSTEM_CHECKS=["1_7.W001"],
    )

class TestMyModel(EvenniaTest):
    def setUp(self):

        if not settings.configured:
           settings.configure(**DEFAULT_SETTINGS)
        django.setup()

        from django.core.management import call_command
        from django.db.models import loading
        loading.cache.loaded = False
        call_command('syncdb', verbosity=0)
```

```
    def tearDown(self):
        settings.configure(**OLD_DEFAULT_SETTINGS)
        django.setup()

        from django.core.management import call_command
        from django.db.models import loading
        loading.cache.loaded = False
        call_command('syncdb', verbosity=0)

    # test cases below ...

    def test_case(self):
        # test case here
```

A note on adding new tests

Having an extensive tests suite is very important for avoiding code degradation as Evennia is developed. Only a small fraction of the Evennia codebase is covered by test suites at this point. Writing new tests is not hard, it's more a matter of finding the time to do so. So adding new tests is really an area where everyone can contribute, also with only limited Python skills.

Testing for Game development (mini-tutorial)

Unit testing can be of paramount importance to game developers. When starting with a new game, it is recommended to look into unit testing as soon as possible; an already huge game is much harder to write tests for. The benefits of testing a game aren't different from the ones regarding library testing. For example it is easy to introduce bugs that affect previously working code. Testing is there to ensure your project behaves the way it should and continue to do so.

If you have never used unit testing (with Python or another language), you might want to check the official Python documentation about unit testing, particularly the first section dedicated to a basic example.

Basic testing using Evennia

Evennia's test runner can be used to launch tests in your game directory (let's call it 'mygame'). Evennia's test runner does a few useful things beyond the normal Python unittest module:

- It creates and sets up an empty database, with some useful objects (players, characters and rooms, among others).
- It provides simple ways to test commands, which can be somewhat tricky at times, if not tested properly.

Therefore, you should use the command-line to execute the test runner, while specifying your own game directories (not the one containing evennia). Go to your game directory (referred as 'mygame' in this section) and execute the test runner:

```
evennia --settings settings.py test commands
```

This command will execute Evennia's test runner using your own settings file. It will set up a dummy database of your choice and look into the 'commands' package defined in your game directory (`mygame/commands` in this example) to find tests. The test module's name should begin with 'test' and contain one or more `TestCase`. A full example can be found below.

A simple example

In your game directory, go to `commands` and create a new file `tests.py` inside (it could be named anything starting with `test`). We will start by making a test that has nothing to do with Commands, just to show how unit testing works:

```
# mygame/commands/tests.py

import unittest

class TestString(unittest.TestCase):

    """Unittest for strings (just a basic example)."""

    def test_upper(self):
        """Test the upper() str method."""
        self.assertEqual('foo'.upper(), 'FOO')
```

This example, inspired from the Python documentation, is used to test the 'upper()' method of the 'str' class. Not very useful, but it should give you a basic idea of how tests are used.

Let's execute that test to see if it works.

```
> evennia --settings settings.py test commands

TESTING: Using specified settings file 'server.conf.settings'.

(Obs: Evennia's full test suite may not pass if the settings are very
different from the default. Use 'test .' as arguments to run only tests
on the game dir.)

Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

We specified the `commands` package to the evennia test command since that's where we put out test file. In this case we could just as well just said `.` to search all of `mygame` for testing files. If we have a lot of tests it may be useful to test only a single set at a time though. We get an information text telling us we are using our custom settings file (instead of Evennia's default file) and then the test runs. The test passes! Change the "FOO" string to something else in the test to see how it looks when it fails.

Testing commands

This section will test the proper execution of the 'abilities' command, as described in the First Steps Coding page. Follow this tutorial to create the 'abilities' command, we will need it to test it.

Testing commands in Evennia is a bit more complex than the simple testing example we have seen. Luckily, Evennia supplies a special test class to do just that ... we just need to inherit from it and use it properly. This class is called 'CommandTest' and is defined in the 'evennia.commands.default.tests' package. To create a test for our 'abilities' command, we just need to create a class that inherits from 'CommandTest' and add methods.

We could create a new test file for this but for now we just append to the `tests.py` file we already have in `commands` from before.

```
# bottom of mygame/commands/tests.py

from evennia.commands.default.tests import CommandTest

from commands.command import CmdAbilities
from typeclasses.characters import Character

class TestAbilities(CommandTest):

    character_typeclass = Character

    def test_simple(self):
        self.call(CmdAbilities(), "", "STR: 5, AGI: 4, MAG: 2")
```

- Line 1-4: we do some importing. 'CommandTest' is going to be our base class for our test, so we need it. We also import our command ('CmdAbilities' in this case). Finally we import the 'Character' typeclass. We need it, since 'CommandTest' doesn't use 'Character', but 'DefaultCharacter', which means the character calling the command won't have the abilities we have written in the 'Character' typeclass.
- Line 6-8: that's the body of our test. Here, a single command is tested in an entire class. Default commands are usually grouped by category in a single class. There is no rule, as long as you know where you put your tests. Note that we set the 'character_typeclass' class attribute to Character. As explained above, if you didn't do that, the system would create a 'DefaultCharacter' object, not a 'Character'. You can try to remove line 4 and 8 to see what happens when running the test.
- Line 10-11: our unique testing method. Note its name: it should begin by 'test_'. Apart from that, the method is quite simple: it's an instance method (so it takes the 'self' argument) but no other arguments is needed. The line 11 uses the 'call' method, which is defined in 'CommandTest'. It's a useful method that compares a command against an expected result. It would be like comparing two strings with 'assertEqual', but the 'call' method does more things, including testing the command in a realistic way (calling its hooks in the right order, so you don't have to worry about that).

Our line 11 could be understood as: test the 'abilities' command (first parameter), with no argument (second parameter), and check that the character using it receives his/her abilities (third parameter).

Let's run our new test:

```
> evennia test unit
[...]
Creating test database for alias 'default'...
..
----------------------------------------------------------------------
Ran 2 tests in 0.156s

OK
Destroying test database for alias 'default'...
```

Two tests were executed, since we have kept 'TestString' from last time. In case of failure, you will get much more information to help you fix the bug.

Coding Utils

Evennia comes with many utilities to help with common coding tasks. Most are accessible directly from the flat API, otherwise you can find them in the `evennia/utils/` folder.

Searching

A common thing to do is to search for objects. The most common time one needs to do this is inside a command body. There it's easiest to use the `search` method defined on all objects. This will search for objects in the same location and inside the caller:

```
obj = self.caller.search(objname)
```

Give the keyword `global_search=True` to extend search to encompass entire database. Also aliases with be matched by this search. You will find multiple examples of this functionality in the default command set.

If you need to search for objects in a code module you can use the functions in `evennia.utils.search`. You can access these as shortcuts `evennia.search_*`.

```
from evennia import search_object
obj = search_object(objname)
```

Note that these latter methods will always return a `list` of results, even if the list has one or zero entries.

Create

Apart from the in-game build commands (`@create` etc), you can also build all of Evennia's game entities directly in code (for example when defining new create commands).

```
import evennia
#
myobj = evennia.create_objects("game.gamesrc.objects.myobj.MyObj",
                               key="MyObj")
myscr = evennia.create_script("game.gamesrc.scripts.myscripts.MyScript",
                              obj=myobj)
helpentry = evennia.create_help_entry("Emoting", "Emoting means that ...")
msg = evennia.create_message(senderobj, [receiverobj], "Hello ...")
channel = evennia.create_channel("news")
player = evennia.create_player("Henry", "henry@test.com", "H@passwd")
```

Each of these create-functions have a host of arguments to further customize the created entity. See `evennia/utils/create.py` for more information.

Logging

Normally you can use Python `print` statements to see output to the terminal/log. The `print` statement should only be used for debugging though. For producion output, use the `logger` which will create proper logs either to terminal or to file.

```
from evennia import logger
#
logger.log_err("This is an Error!")
logger.log_warn("This is a Warning!")
logger.log_info("This is normal information")
logger.log_dep("This feature is deprecated")
```

There is a special log-message type, `log_trace()` that is intended to be called from inside a traceback - this can be very useful for relaying the traceback message back to log without having it kill the server.

```
try:
  # [some code that may fail...]
except Exception:
  logger.log_trace("This text will show beneath the traceback itself.")
```

The `log_file` logger, finally, is a very useful logger for outputting arbitrary log messages. This is a heavily optimized asynchronous log mechanism using threads to avoid overhead. You should be able to use it for very heavy custom logging without fearing disk-write delays.

```
logger.log_file(message, filename="mylog.log")
```

If not an absolute path is given, the log file will appear in the `mygame/server/logs/` directory. If the file already exists, it will be appended to. Timestamps on the same format as the normal Evennia logs will be automatically added to each entry. If a filename is not specified, output will be written to a file `game/logs/game.log`.

Game time

Evennia tracks the current server time. You can access this time via the `evennia.gametime` shortcut:

```
from evennia import gametime

# all the functions below return times in seconds).

# total running time of the server
runtime = gametime.runtime()
# time since latest hard reboot (not including reloads)
uptime = gametime.uptime()
# server epoch (its start time)
server_epoch = gametime.server_epoch()

# in-game epoch (this can be set by `settings.TIME_GAME_EPOCH`.
# If not, the server epoch is used.
game_epoch = gametime.game_epoch()
# in-game time passed since time started running
gametime = gametime.gametime()
# in-game time plus game epoch (i.e. the current in-game
# time stamp)
gametime = gametime.gametime(absolute=True)
# reset the game time (back to game epoch)
gametime.reset_gametime()
```

The setting `TIME_FACTOR` determines how fast/slow in-game time runs compared to the real world. The setting `TIME_GAME_EPOCH` sets the starting game epoch (in seconds). The functions from the `gametime` module all return their times in seconds. You can convert this to whatever units of time you desire for your game. You can use the `@time` command to view the server time info.

You can also *schedule* things to happen at specific in-game times using the gametime.schedule function:

```
import evennia

def church_clock:
    limbo = evennia.search_object(key="Limbo")
    limbo.msg_contents("The church clock chimes two.")

gametime.schedule(church_clock, hour=2)
```

utils.time_format()

This function takes a number of seconds as input (e.g. from the `gametime` module above) and converts it to a nice text output in days, hours etc. It's useful when you want to show how old something is. It converts to four different styles of output using the *style* keyword:

- style 0 - `5d:45m:12s` (standard colon output)
- style 1 - `5d` (shows only the longest time unit)
- style 2 - `5 days, 45 minutes` (full format, ignores seconds)
- style 3 - `5 days, 45 minutes, 12 seconds` (full format, with seconds)

utils.inherits_from()

This useful function takes two arguments - an object to check and a parent. It returns `True` if object inherits from parent *at any distance* (as opposed to Python's in-built `is_instance()` that will only catch immediate dependence). This function also accepts as input any combination of classes, instances or python-paths-to-classes.

Note that Python code should usually work with duck typing. But in Evennia's case it can sometimes be useful to check if an object inherits from a given Typeclass as a way of identification. Say for example that we have a typeclass *Animal*. This has a subclass *Felines* which in turn has a subclass *HouseCat*. Maybe there are a bunch of other animal types too, like horses and dogs. Using `inherits_from` will allow you to check for all animals in one go:

```
from evennia import utils
if (utils.inherits_from(obj, "typeclasses.objects.animals.Animal"):
   obj.msg("The bouncer stops you in the door. He says: 'No talking animals allowed.'
↪")
```

utils.delay()

This is a thin wrapper around a Twisted construct called a *deferred*. It simply won't return until a given number of seconds have passed, at which time it will trigger a given callback with whatever argument. This is a small and lightweight (non-persistent) alternative to a full Scripts. Contrary to a Script it can also handle sub-second timing precision (although this is not something you should normally need to worry about).

Some text utilities

In a text game, you are naturally doing a lot of work shuffling text back and forth. Here is a *non-complete* selection of text utilities found in `evennia/utils/utils.py` (shortcut `evennia.utils`). If nothing else it can be good to look here before starting to develop a solution of your own.

utils.fill()

This flood-fills a text to a given width (shuffles the words to make each line evenly wide). It also indents as needed.

```
outtxt = fill(intxt, width=78, indent=4)
```

utils.crop()

This function will crop a very long line, adding a suffix to show the line actually continues. This can be useful in listings when showing multiple lines would mess up things.

```
intxt = "This is a long text that we want to crop."
outtxt = crop(intxt, width=19, suffix="[...]")
# outtxt is now "This is a long text[...]"
```

utils.dedent()

This solves what may at first glance appear to be a trivial problem with text - removing indentations. It is used to shift entire paragraphs to the left, without disturbing any further formatting they may have. A common case for this is when using Python triple-quoted strings in code - they will retain whichever indentation they have in the code, and to make easily-readable source code one usually don't want to shift the string to the left edge.

```
#python code is entered at a given indentation
      intxt = """
      This is an example text that will end
      up with a lot of whitespace on the left.
                It also has indentations of
                its own."""
      outtxt = dedent(intxt)
      # outtxt will now retain all internal indentation
      # but be shifted all the way to the left.
```

Normally you do the dedent in the display code (this is for example how the help system homogenizes help entries).

text conversion()

Evennia supplies two utility functions for converting text to the correct encodings. `to_str()` and `to_unicode()`. The difference from Python's in-built `str()` and `unicode()` operators are that the Evennia ones makes use of the `ENCODINGS` setting and will try very hard to never raise a traceback but instead echo errors through logging. See here for more info.

Making ascii tables

The EvTable class (`evennia/utils/evtable.py`) can be used to create correctly formatted text tables. There is also EvForm (`evennia/utils/evform.py`). This reads a fixed-format text template from a file in order to create any level of sophisticated ascii layout. Both evtable and evform have lots of options and inputs so see the header of each module for help.

The third-party PrettyTable module is also included in Evennia. PrettyTable is considered deprecated in favor of EvTable since PrettyTable cannot handle ANSI colour. PrettyTable can be found in `evennia/utils/prettytable/`. See its homepage above for instructions.

Async Process

This is considered an advanced topic.

Synchronous versus Asynchronous

Most program code operates *synchronously*. This means that each statement in your code gets processed and finishes before the next can begin. This makes for easy-to-understand code. It is also a *requirement* in many cases - a subsequent piece of code often depend on something calculated or defined in a previous statement.

Consider this piece of code:

```
print "before call ..."
long_running_function()
print "after call ..."
```

When run, this will print `"before call ..."`, after which the `long_running_function` gets to work for however long time. Only once that is done, the system prints `"after call ..."`. Easy and logical to follow. Most of Evennia work in this way and often it's important that commands get executed in the same strict order they were coded.

Evennia, via Twisted, is a single-process multi-user server. In simple terms this means that it swiftly switches between dealing with player input so quickly that each player feels like they do things at the same time. This is a clever illusion however: If one user, say, runs a command containing that `long_running_function`, *all* other players are effectively forced to wait until it finishes.

Now, it should be said that on a modern computer system this is rarely an issue. Very few commands run so long that other users notice it. And as mentioned, most of the time you *want* to enforce all commands to occur in strict sequence.

When delays do become noticeable and you don't care in which order the command actually completes, you can run it *asynchronously*. This makes use of the `run_async()` function in `src/utils/utils.py`:

```
run_async(function, *args, **kwargs)
```

Where `function` will be called asynchronously with `*args` and `**kwargs`. Example:

```
from evevnnia import utils
print "before call ..."
utils.run_async(long_running_function)
print "after call ..."
```

Now, when running this you will find that the program will not wait around for `long_running_function` to finish. In fact you will see `"before call ..."` and `"after call ..."` printed out right away. The long-running function will run in the background and you (and other users) can go on as normal.

Customizing asynchronous operation

A complication with using asynchronous calls is what to do with the result from that call. What if `long_running_function` returns a value that you need? It makes no real sense to put any lines of code after the call to try to deal with the result from `long_running_function` above - as we saw the `"after call ..."` got printed long before `long_running_function` was finished, making that line quite pointless for processing any data from the function. Instead one has to use *callbacks*.

`utils.run_async` takes reserved kwargs that won't be passed into the long-running function:

- `at_return(r)` (the *callback*) is called when the asynchronous function (`long_running_function` above) finishes successfully. The argument `r` will then be the return value of that function (or `None`).

  ```
  def at_return(r):
      print r
  ```

- `at_return_kwargs` - an optional dictionary that will be fed as keyword arguments to the `at_return` callback.
- `at_err(e)` (the *errback*) is called if the asynchronous function fails and raises an exception. This exception is passed to the errback wrapped in a *Failure* object `e`. If you do not supply an errback of your own, Evennia will automatically add one that silently writes errors to the evennia log. An example of an errback is found below:

```
def at_err(e):
    print "There was an error:", str(e)
```

- `at_err_kwargs` - an optional dictionary that will be fed as keyword arguments to the `at_err` errback.

An example of making an asynchronous call from inside a Command definition:

```
from ev import utils
from game.gamesrc.commands.basecommand import Command

class CmdAsync(Command):

   key = "asynccommand"

   def func(self):

       def long_running_function():
           #[... lots of time-consuming code
           return final_value

       def at_return(r):
           self.caller.msg("The final value is %s" % r)

       def at_err(e):
           self.caller.msg("There was an error: %s" % e)

       # do the async call, setting all callbacks
       utils.run_async(long_running_function, at_return, at_err)
```

That's it - from here on we can forget about `long_running_function` and go on with what else need to be done. *Whenever* it finishes, the `at_return` function will be called and the final value will pop up for us to see. If not we will see an error message.

Process Pool

> Note: The Process pool is currently not available nor supported, so the following section should be ignored. An old and incompatible version of the procpool can be found in the evennia/procpool repository if you are interested.

The `ProcPool` is an Evennia subsystem that launches a pool of processes based on the ampoule package (included with Evennia). When active, `run_async` will use this pool to offload its commands. `ProcPool` is deactivated by default, it can be turned on with `settings.PROCPOOL_ENABLED`. *It should be noted that the default SQLite3 database is not suitable for for multiprocess operation. So if you use ''ProcPool'' you should consider switching to another database such as MySQL or PostgreSQL.*

The Process Pool makes several additional options available to `run_async`. The following keyword arguments make sense when `ProcPool` is active:

- `use_thread` - this force-reverts back to thread operation (as above). It effectively deactivates all additional features `ProcPool` offers.

- `proc_timeout` - this enforces a timeout for the running process in seconds; after this time the process will be killed.
- `at_return`, `at_err` - these work the same as above.

In addition to feeding a single callable to `run_async`, the first argument may also be a source string. This is a piece of python source code that will be executed in a subprocess via `ProcPool`. Any extra keyword arguments to `run_async` that are not one of the reserved ones will be used to specify what will be available in the execution environment.

There is one special variable used in the remote execution: `_return`. This is a function, and all data fed to `_return` will be returned from the execution environment and appear as input to your `at_return` callback (if it is defined). You can call `_return` multiple times in your code - the return value will then be a list.

Example:

```
from src.utils.utils import run_async

source = """
from time import sleep
sleep(5) # sleep five secs
val = testvar + 5
_return(val)
_return(val + 5)
"""

# we assume myobj is a character retrieved earlier
# these callbacks will just print results/errors
def callback(ret):
    myobj.msg(ret)
def errback(err):
    myobj.msg(err)
testvar = 3

# run async
run_async(source, at_return=callback, at_err=errback, testvar=testvar)

# this will return '[8, 13]'
```

You can also test the async mechanism from in-game using the `@py` command:

```
@py from src.utils.utils import run_async;run_async("_return(1+2)",at_return=self.msg)
```

Note: The code execution runs without any security checks, so it should not be available to unprivileged users. Try `contrib.evlang.evlang.limited_exec` for running a more restricted version of Python for untrusted users. This will use `run_async` under the hood.

delay

The `delay` function is a much simpler sibling to `run_async`. It is in fact just a way to delay the execution of a command until a future time. This is equivalent to something like `time.sleep()` except delay is asynchronous while `sleep` would lock the entire server for the duration of the sleep.

```
def callback(obj):
   obj.msg("Returning!")
delay(10, caller, callback=callback)
```

This will delay the execution of the callback for 10 seconds. This function is explored much more in the Command Duration Tutorial.

Assorted notes

Note that the `run_async` will try to launch a separate thread behind the scenes. Some databases, notably our default database SQLite3, does *not* allow concurrent read/writes. So if you do a lot of database access (like saving to an Attribute) in your function, your code might actually run *slower* using this functionality if you are not careful. Extensive real-world testing is your friend here.

Overall, be careful with choosing when to use asynchronous calls. It is mainly useful for large administration operations that have no direct influence on the game world (imports and backup operations come to mind). Since there is no telling exactly when an asynchronous call actually ends, using them for in-game commands is to potentially invite confusion and inconsistencies (and very hard-to-reproduce bugs).

The very first synchronous example above is not *really* correct in the case of Twisted, which is inherently an asynchronous server. Notably you might find that you will *not* see the first `before call ...` text being printed out right away. Instead all texts could end up being delayed until after the long-running process finishes. So all commands will retain their relative order as expected, but they may appear with delays or in groups.

Further reading

Technically, `run_async` is just a very thin and simplified wrapper around a Twisted Deferred object; the wrapper sets up a separate thread and assigns a default errback also if none is supplied. If you know what you are doing there is nothing stopping you from bypassing the utility function, building a more sophisticated callback chain after your own liking.

CHAPTER 13

Tutorials

This chapter holds tutorials and step-by-step instructions on various Evennia topics.

First Steps Coding

This section gives a brief step-by-step introduction on how to set up Evennia for the first time so you can modify and overload the defaults easily. You should only need to do these steps once. It also walks through you making your first few tweaks.

Before continuing, make sure you have Evennia installed and running by following the Getting Started instructions. You should have initialized a new game folder with the `evennia --init foldername` command. We will in the following assume this folder is called "mygame".

It might be a good idea to eye through the brief Coding Introduction too (especially the recommendations in the section about the evennia "flat" API will help you here and in the future).

To follow this tutorial you also need to know the basics of operating your computer's terminal/command line. You also need to have a text editor to edit and create source text files. There are plenty of online tutorials on how to use the terminal and plenty of good free text editors. We will assume these things are already familiar to you henceforth.

Your first changes

Below are some first things to try with your new custom modules. You can test these to get a feel for the system. See also Tutorials for more step-by-step help and special cases.

Tweak default Character

We will add some simple rpg attributes to our default Character. In the next section we will follow up with a new command to view those attributes.

1. Edit `mygame/typeclasses/characters.py` and modify the `Character` class. The `at_object_creation` method also exists on the `DefaultCharacter` parent and will overload it. The `get_abilities` method is unique to our version of `Character`.

```
class Character(DefaultCharacter):
    # [...]
    def at_object_creation(self):
        """
        Called only at initial creation. This is a rather silly
        example since ability scores should vary from Character to
        Character and is usually set during some character
        generation step instead.
        """
        #set persistent attributes
        self.db.strength = 5
        self.db.agility = 4
        self.db.magic = 2

    def get_abilities(self):
        """
        Simple access method to return ability
        scores as a tuple (str,agi,mag)
        """
        return self.db.strength, self.db.agility, self.db.magic
```

2. Reload the server (you will still be connected to the game after doing this).

Updating yourself

Note that the new Attributes will only be stored on *newly* created characters (`at_object_creation` is only called when the object is first created). So if you call the `get_abilities` hook on yourself at this point you will see the Attribute have not been set:

```
# (you have to be superuser to use @py)
@py self.get_abilities()
<<< (None, None, None)
```

This is because your Character was already created before you made your changes to the `Character` class and thus the `at_object_creation()` hook will not be called again.

This is easily remedied though - you can force re-run the startup hooks on yourself with the `@typeclass` command:

```
@typeclass/force self
```

This will re-run `at_object_creation` on yourself (in code you can use the `Character.swap_typeclass` method with the same typeclass set). You should henceforth be able to get the abilities successfully:

```
@py self.get_abilities()
<<< (5, 4, 2)
```

See the Object Typeclass tutorial for more help and the Typeclasses and Attributes page for detailed documentation about Typeclasses and Attributes.

Trouble Shooting: Updating yourself

One may experience errors for a number of reasons. Common beginner errors are spelling mistakes, wrong indentations or code omissions leading to a `SyntaxError`. Let's say you leave out a colon from the end of a class function like so: `def at_object_creation(self)`. The client will reload without issue. *However*, if you look at the terminal/console (i.e. not in-game), you will see Evennia complaining (this is called a *traceback*):

```
Traceback (most recent call last):
File "C:\mygame\typeclasses\characters.py", line 33
     def at_object_creation(self)
                                 ^
SyntaxError: invalid syntax
```

Evennia will still be restarting and following the tutorial, doing `@py self.get_abilities()` will return the right response `(None, None, None)`. But when attempting to `@typeclass/force self` you will get this response:

```
AttributeError: 'DefaultObject' object has no attribute 'get_abilities'
```

The full error will show in the terminal/console but this is confusing since you did add `get_abilities` before. Note however what the error says - you (`self`) should be a `Character` but the error talks about `DefaultObject`. What has happened is that due to your unhandled `SyntaxError` earlier, Evennia could not load the `character.py` module at all (it's not valid Python). Rather than crashing, Evennia handles this by temporarily falling back to a safe default - `DefaultObject` - in order to keep your MUD running. Fix the original `SyntaxError` and reload the server. Evennia will then be able to use your modified `Character` class again and things should work.

> Note: Learning how to interpret an error traceback is a critical skill for anyone learning Python. Full tracebacks will appear in the terminal/Console you started Evennia from. The traceback text can sometimes be quite long, but you are usually just looking for the last few lines: The description of the error and the filename + line number for where the error occurred. In the example above, we see it's a `SyntaxError` happening at `line 33` of `mygame\typeclasses\characters.py`. In this case it even points out *where* on the line it encountered the error (the missing colon). Learn to read tracebacks and you'll be able to resolve the vast majority of common errors easily.

Add a new default command

The `@py` command used above is only available to privileged users. We want any player to be able to see their stats. Let's add a new command to list the abilities we added in the previous section.

1. Open `mygame/commands/command.py`. You could in principle put your command anywhere but this module has all the imports already set up along with some useful documentation. Make a new class at the bottom of this file:

```
class CmdAbilities(Command):
    """
    List abilities

    Usage:
      abilities

    Displays a list of your current ability values.
    """
    key = "abilities"
    aliases = ["abi"]
    lock = "cmd:all()"
    help_category = "General"
```

```
    def func(self):
        "implements the actual functionality"

         str, agi, mag = self.caller.get_abilities()
         string = "STR: %s, AGI: %s, MAG: %s" % (str, agi, mag)
         self.caller.msg(string)
```

2. Next you edit `mygame/commands/default_cmdsets.py` and add a new import to it near the top:

```
from commands.command import CmdAbilities
```

3. In the `CharacterCmdSet` class, add the following near the bottom (it says where):

```
self.add(CmdAbilities())
```

4. Reload the server (noone will be disconnected by doing this).

You (and anyone else) should now be able to use `abilities` (or its alias `abi`) as part of your normal commands in-game:

```
abilities
STR: 5, AGI: 4, MAG: 2
```

See the Adding a Command tutorial for more examples and the Commands section for detailed documentation about the Command system.

Make a new type of object

Let's test to make a new type of object. This example is an "wise stone" object that returns some random comment when you look at it, like this:

```
> look stone

A very wise stone

This is a very wise old stone.
It grumbles and says: 'The world is like a rock of chocolate.'
```

1. Create a new module in `mygame/typeclasses/`. Name it `wiseobject.py` for this example.
2. In the module import the base `Object` (`typeclasses.objects.Object`). This is empty by default, meaning it is just a proxy for the default `evennia.DefaultObject`.
3. Make a new class in your module inheriting from `Object`. Overload hooks on it to add new functionality. Here is an example of how the file could look:

```
from random import choice
from typeclasses.objects import Object

class WiseObject(Object):
    """
    An object speaking when someone looks at it. We
    assume it looks like a stone in this example.
    """
    def at_object_creation(self):
        "Called when object is first created"
```

```
        self.db.wise_texts = \
               ["Stones have feelings too.",
                "To live like a stone is to not have lived at all.",
                "The world is like a rock of chocolate."]

    def return_appearance(self, looker):
        """
        Called by the look command. We want to return
        a wisdom when we get looked at.
        """
        # first get the base string from the
        # parent's return_appearance.
        string = super(WiseObject, self).return_appearance(looker)
        wisewords = "\n\nIt grumbles and says: '%s'"
        wisewords = wisewords % choice(self.db.wise_texts)
        return string + wisewords
```

4. Check your code for bugs. Tracebacks will appear on your command line or log. If you have a grave Syntax Error in your code, the source file itself will fail to load which can cause issues with the entire cmdset. If so, fix your bug and reload the server from the command line (noone will be disconnected by doing this).
5. Use `@create/drop stone:wiseobject.WiseObject` to create a talkative stone. If the `@create` command spits out a warning or cannot find the typeclass (it will tell you which paths it searched), re-check your code for bugs and that you gave the correct path. The `@create` command starts looking for Typeclasses in `mygame/typeclasses/`.
6. Use `look stone` to test. You will see the default description ("You see nothing special") followed by a random message of stony wisdom. Use `@desc stone = This is a wise old stone.` to make it look nicer. See the Builder Docs for more information.

Note that `at_object_creation` is only called once, when the stone is first created. If you make changes to this method later, already existing stones will not see those changes. As with the `Character` example above you can use `@typeclass/force` to tell the stone to re-run its initialization.

The `at_object_creation` is a special case though. Changing most other aspects of the typeclass does *not* require manual updating like this - you just need to `@reload` to have all changes applied automatically to all existing objects.

Where to go from here?

There are more Tutorials, including one for building a whole little MUSH-like game - that is instructive also if you have no interest in MUSHes per se. A good idea is to also get onto the IRC chat and the mailing list to get in touch with the community and other developers.

Tutorial for basic MUSH like game

This tutorial lets you code a small but complete and functioning MUSH-like game in Evennia. A MUSH is, for our purposes, a class of roleplay-centric games focused on free form storytelling. Even if you are not interested in MUSH:es, this is still a good first game-type to try since it's not so code heavy. You will be able to use the same principles for building other types of games.

The tutorial starts from scratch. If you did the First Steps Coding tutorial already you should have some ideas about how to do some of the steps already.

The following are the (very simplistic and cut-down) features we will implement (this was taken from a feature request from a MUSH user new to Evennia). A Character in this system should:

- Have a "Power" score from 1 to 10 that measures how strong they are (stand-in for the stat system).
- Have a command (e.g. `+setpower 4`) that sets their power (stand-in for character generation code).
- Have a command (e.g. `+attack`) that lets them roll their power and produce a "Combat Score" between `1` and `10*Power`, displaying the result and editing their object to record this number (stand-in for `+actions` in the command code).
- Have a command that displays everyone in the room and what their most recent "Combat Score" roll was (stand-in for the combat code).
- Have a command (e.g. `+createNPC Jenkins`) that creates an NPC with full abilities.
- Have a command to control NPCs, such as `+npc/cmd (name)=(command)` (stand-in for the NPC controlling code).

In this tutorial we will assume you are starting from an empty database without any previous modifications.

Server Settings

To emulate a MUSH, the default `MULTISESSION_MODE=0` is enough (one unique session per player/character). This is the default so you don't need to change anything. You will still be able to puppet/unpuppet objects you have permission to, but there is no character selection out of the box in this mode.

We will assume our game folder is called `mygame` henceforth. You should be fine with the default SQLite3 database.

Creating the Character

First thing is to choose how our Character class works. We don't need to define a special NPC object – an NPC is after all just a Character without a Player currently controlling them.

Make your changes in the `mygame/typeclasses/characters.py` file:

```
# mygame/typeclasses/characters.py

from evennia import DefaultCharacter

class Character(DefaultCharacter):
    """
     [...]
    """
    def at_object_creation(self):
        "This is called when object is first created, only."
        self.db.power = 1
        self.db.combat_score = 1
```

We defined two new Attributes `power` and `combat_score` and set them to default values. Make sure to `@reload` the server if you had it already running (you need to reload every time you update your python code, don't worry, no players will be disconnected by the reload).

Note that only *new* characters will see your new Attributes (since the `at_object_creation` hook is called when the object is first created, existing Characters won't have it). To update yourself, run

```
@typeclass/force self
```

This resets your own typeclass (the `/force` switch is a safety measure to not do this accidentally), this means that `at_object_creation` is re-run.

```
examine self
```

Under the "Persistent attributes" heading you should now find the new Attributes `power` and `score` set on yourself by `at_object_creation`. If you don't, first make sure you `@reload`ed into the new code, next look at your server log (in the terminal/console) to see if there were any syntax errors in your code that may have stopped your new code from loading correctly.

Character Generation

We assume in this example that Players first connect into a "character generation area". Evennia also supports full OOC menu-driven character generation, but for this example, a simple start room is enough. When in this room (or rooms) we allow character generation commands. In fact, character generation commands will *only* be available in such rooms.

Note that this again is made so as to be easy to expand to a full-fledged game. With our simple example, we could simply set an `is_in_chargen` flag on the player and have the `+setpower` command check it. Using this method however will make it easy to add more functionality later.

What we need are the following:

- One character generation Command to set the "Power" on the `Character`.
- A chargen CmdSet to hold this command. Lets call it `ChargenCmdset`.
- A custom `ChargenRoom` type that makes this set of commands available to players in such rooms.
- One such room to test things in.

The +setpower command

For this tutorial we will add all our new commands to `mygame/commands/command.py` but you could split your commands into multiple module if you prefered.

For this tutorial character generation will only consist of one Command to set the Character s "power" stat. It will be called on the following MUSH-like form:

```
+setpower 4
```

Open `command.py` file. It contains documented empty templates for the base command and the "MuxCommand" type used by default in Evennia. We will use the plain `Command` type here, the `MuxCommand` class offers some extra features like stripping whitespace that may be useful - if so, just import from that instead.

Add the following to the end of the `command.py` file:

```
# end of command.py
from evennia import Command # just for clarity; already imported above

class CmdSetPower(Command):
    """
    set the power of a character

    Usage:
      +setpower <1-10>

    This sets the power of the current character. This can only be
    used during character generation.
    """
```

```
    key = "+setpower"
    help_category = "mush"

    def func(self):
        "This performs the actual command"
        errmsg = "You must supply a number between 1 and 10."
        if not self.args:
            self.caller.msg(errmsg)
            return
        try:
            power = int(self.args)
        except ValueError:
            self.caller.msg(errmsg)
            return
        if not (1 <= power <= 10):
            self.caller.msg(errmsg)
            return
        # at this point the argument is tested as valid. Let's set it.
        self.caller.db.power = power
        self.caller.msg("Your Power was set to %i." % power)
```

This is a pretty straightforward command. We do some error checking, then set the power on ourself. We use a `help_category` of "mush" for all our commands, just so they are easy to find and separate in the help list.

Save the file. We will now add it to a new CmdSet so it can be accessed (in a full chargen system you would of course have more than one command here).

Open `mygame/commands/default_cmdsets.py` and import your `command.py` module at the top. We also import the default `CmdSet` class for the next step:

```
from evennia import CmdSet
from commands import command
```

Next scroll down and define a new command set (based on the base `CmdSet` class we just imported at the end of this file, to hold only our chargen-specific command(s):

```
# end of default_cmdsets.py

class ChargenCmdset(CmdSet):
    """
    This cmdset it used in character generation areas.
    """
    key = "Chargen"
    def at_cmdset_creation(self):
        "This is called at initialization"
        self.add(command.CmdSetPower())
```

In the future you can add any number of commands to this cmdset, to expand your character generation system as you desire. Now we need to actually put that cmdset on something so it's made available to users. We could put it directly on the Character, but that would make it available all the time. It's cleaner to put it on a room, so it's only available when players are in that room.

Chargen areas

We will create a simple Room typeclass to act as a template for all our Chargen areas. Edit `mygame/typeclasses/rooms.py` next:

```
# end of rooms.py

from commands.default_cmdsets import ChargenCmdset

class ChargenRoom(Room):
    """
    This room class is used by character-generation rooms. It makes
    the ChargenCmdset available.
    """
    def at_object_creation(self):
        "this is called only at first creation"
        self.cmdset.add(ChargenCmdset, permanent=True)
```

Note how new rooms created with this typeclass will always start with `ChargenCmdset` on themselves. Don't forget the `permanent=True` keyword or you will loose the cmdset after a server reload. For more information about Command Sets and Commands, see the respective links.

Testing chargen

First, make sure you have `@reload`ed the server (or use `evennia reload` from the terminal) to have your new python code added to the game. Check your terminal and fix any errors you see - the error traceback lists exactly where the error is found - look line numbers in files you have changed.

We can't test things unless we have some chargen areas to test. Log into the game (you should at this point be using the new, custom Character class). Let's dig a chargen area to test.

```
@dig chargen:rooms.ChargenRoom = chargen,finish
```

If you read the help for `@dig` you will find that this will create a new room named `chargen`. The part after the `:` is the python-path to the Typeclass you want to use. Since Evennia will automatically try the `typeclasses` folder of our game directory, we just specify `rooms.ChargenRoom`, meaning it will look inside the module `rooms.py` for a class named `ChargenRoom` (which is what we created above). The names given after `=` are the names of exits to and from the room from your current location. You could also append aliases to each one name, such as `chargen;character generation`.

So in summary, this will create a new room of type ChargenRoom and open an exit `chargen` to it and an exit back here named `finish`. If you see errors at this stage, you must fix them in your code. `@reload` between fixes. Don't continue until the creation seems to have worked okay.

```
chargen
```

This should bring you to the chargen room. Being in there you should now have the `+setpower` command available, so test it out. When you leave (via the `finish` exit), the command will go away and trying `+setpower` should now give you a command-not-found error. Use `ex me` (as a privileged user) to check so the `Power` Attribute has been set correctly.

If things are not working, make sure your typeclasses and commands are free of bugs and that you have entered the paths to the various command sets and commands correctly. Check the logs or command line for tracebacks and errors.

Combat System

We will add our combat command to the default command set, meaning it will be available to everyone at all times. The combat system consists of a `+attack` command to get how successful our attack is. We also change the default `look` command to display the current combat score.

Attacking with the +attack command

Attacking in this simple system means rolling a random "combat score" influenced by the `power` stat set during Character generation:

```
> +attack
You +attack with a combat score of 12!
```

Go back to `mygame/commands/command.py` and add the command to the end like this:

```
import random

class CmdAttack(Command):
    """
    issues an attack

    Usage:
        +attack

    This will calculate a new combat score based on your Power.
    Your combat score is visible to everyone in the same location.
    """
    key = "+attack"
    help_category = "mush"

    def func(self):
        "Calculate the random score between 1-10*Power"
        caller = self.caller
        power = caller.db.power
        if not power:
            # this can happen if caller is not of
            # our custom Character typeclass
            power = 1
        combat_score = random.randint(1, 10 * power)
        caller.db.combat_score = combat_score

        # announce
        message = "%s +attack%s with a combat score of %s!"
        caller.msg(message % ("You", "", combat_score))
        caller.location.msg_contents(message %
                                     (caller.key, "s", combat_score),
                                     exclude=caller)
```

What we do here is simply to generate a "combat score" using Python's inbuilt `random.randint()` function. We then store that and echo the result to everyone involved.

To make the `+attack` command available to you in game, go back to `mygame/commands/default_cmdsets.py` and scroll down to the `CharacterCmdSet` class. At the correct place add this line:

```
self.add(command.CmdAttack())
```

`@reload` Evennia and the `+attack` command should be available to you. Run it and use e.g. `@ex` to make sure the `combat_score` attribute is saved correctly.

Have "look" show combat scores

Players should be able to view all current combat scores in the room. We could do this by simply adding a second command named something like `+combatscores`, but we will instead let the default `look` command do the heavy lifting for us and display our scores as part of its normal output, like this:

```
>  look Tom
Tom (combat score: 3)
This is a great warrior.
```

We don't actually have to modify the `look` command itself however. To understand why, take a look at how the default `look` is actually defined. It sits in `evennia/commands/default/general.py` (or browse it online here). You will find that the actual return text is done by the `look` command calling a *hook method* named `return_appearance` on the object looked at. All the `look` does is to echo whatever this hook returns. So what we need to do is to edit our custom Character typeclass and overload its `return_appearance` to return what we want (this is where the advantage of having a custom typeclass comes into play for real).

Go back to your custom Character typeclass in `mygame/typeclasses/characters.py`. The default implementation of `return appearance` is found in `evennia.DefaultCharacter` (or online here). If you want to make bigger changes you could copy & paste the whole default thing into our overloading method. In our case the change is small though:

```
class Character(DefaultCharacter):
    """
     [...]
    """
    def at_object_creation(self):
        "This is called when object is first created, only."
        self.db.power = 1
        self.db.combat_score = 1

    def return_appearance(self, looker):
        """
        The return from this method is what
        looker sees when looking at this object.
        """
        text = super(Character, self).return_appearance(looker)
        cscore = " (combat score: %s)" % self.db.combat_score
        if "\n" in text:
            # text is multi-line, add score after first line
            first_line, rest = text.split("\n", 1)
            text = first_line + cscore + "\n" + rest
        else:
            # text is only one line; add score to end
            text += cscore
        return text
```

What we do is to simply let the default `return_appearance` do its thing (`super` will call the parent's version of the same method). We then split out the first line of this text, append our `combat_score` and put it back together again.

`@reload` the server and you should be able to look at other Characters and see their current combat scores.

> Note: A potentially more useful way to do this would be to overload the entire `return_appearance`

of the Rooms of your mush and change how they list their contents; in that way one could see all combat scores of all present Characters at the same time as looking at the room. We leave this as an exercise.

NPC system

Here we will re-use the Character class by introducing a command that can create NPC objects. We should also be able to set its Power and order it around.

There are a few ways to define the NPC class. We could in theory create a custom typeclass for it and put a custom NPC-specific cmdset on all NPCs. This cmdset could hold all manipulation commands. Since we expect NPC manipulation to be a common occurrence among the user base however, we will instead put all relevant NPC commands in the default command set and limit eventual access with Permissions and Locks.

Creating an NPC with +createNPC

We need a command for creating the NPC, this is a very straightforward command:

```
> +createnpc Anna
You created the NPC 'Anna'.
```

At the end of command.py, create our new command:

```
from evennia import create_object

class CmdCreateNPC(Command):
    """
    create a new npc

    Usage:
        +createNPC <name>

    Creates a new, named NPC. The NPC will start with a Power of 1.
    """
    key = "+createnpc"
    aliases = ["+createNPC"]
    locks = "call:not perm(nonpcs)"
    help_category = "mush"

    def func(self):
        "creates the object and names it"
        caller = self.caller
        if not self.args:
            caller.msg("Usage: +createNPC <name>")
            return
        if not caller.location:
            # may not create npc when OOC
            caller.msg("You must have a location to create an npc.")
            return
        # make name always start with capital letter
        name = self.args.strip().capitalize()
        # create npc in caller's location
        npc = create_object("characters.Character",
                    key=name,
                    location=caller.location,
                    locks="edit:id(%i) and perm(Builders);call:false()" % caller.id)
        # announce
```

```
        message = "%s created the NPC '%s'."
        caller.msg(message % ("You", name))
        caller.location.msg_contents(message % (caller.key, name),
                                             exclude=caller)
```

Here we define a `+createnpc` (`+createNPC` works too) that is callable by everyone *not* having the `nonpcs` "permission" (in Evennia, a "permission" can just as well be used to block access, it depends on the lock we define). We create the NPC object in the caller's current location, using our custom `Character` typeclass to do so.

We set an extra lock condition on the NPC, which we will use to check who may edit the NPC later – we allow the creator to do so, and anyone with the Builders permission (or higher). See Locks for more information about the lock system.

Note that we just give the object default permissions (by not specifying the `permissions` keyword to the `create_object()` call). In some games one might want to give the NPC the same permissions as the Character creating them, this might be a security risk though.

Add this command to your default cmdset the same way you did the `+attack` command earlier. `@reload` and it will be available to test.

Editing the NPC with +editNPC

Since we re-used our custom character typeclass, our new NPC already has a *Power* value - it defaults to 1. How do we change this?

There are a few ways we can do this. The easiest is to remember that the `power` attribute is just a simple Attribute stored on the NPC object. So as a Builder or Admin we could set this right away with the default `@set` command:

```
@set mynpc/power = 6
```

The `@set` command is too generally powerful though, and thus only available to staff. We will add a custom command that only changes the things we want players to be allowed to change. We could in principle re-work our old `+setpower` command, but let's try something more useful. Let's make a `+editNPC` command.

```
> +editNPC Anna/power = 10
Set Anna's property 'power' to 10.
```

This is a slightly more complex command. It goes at the end of your `command.py` file as before.

```
class CmdEditNPC(Command):
    """
    edit an existing NPC

    Usage:
      +editnpc <name>[/<attribute> [= value]]

    Examples:
      +editnpc mynpc/power = 5
      +editnpc mynpc/power    - displays power value
      +editnpc mynpc          - shows all editable
                                attributes and values

    This command edits an existing NPC. You must have
    permission to edit the NPC to use this.
    """
    key = "+editnpc"
    aliases = ["+editNPC"]
```

```
    locks = "cmd:not perm(nonpcs)"
    help_category = "mush"

    def parse(self):
        "We need to do some parsing here"
        args = self.args
        propname, propval = None, None
        if "=" in args:
            args, propval = [part.strip() for part in args.rsplit("=", 1)]
        if "/" in args:
            args, propname = [part.strip() for part in args.rsplit("/", 1)]
        # store, so we can access it below in func()
        self.name = args
        self.propname = propname
        # a propval without a propname is meaningless
        self.propval = propval if propname else None

    def func(self):
        "do the editing"

        allowed_propnames = ("power", "attribute1", "attribute2")

        caller = self.caller
        if not self.args or not self.name:
            caller.msg("Usage: +editnpc name[/propname][=propval]")
            return
        npc = caller.search(self.name)
        if not npc:
            return
        if not npc.access(caller, "edit"):
            caller.msg("You cannot change this NPC.")
            return
        if not self.propname:
            # this means we just list the values
            output = "Properties of %s:" % npc.key
            for propname in allowed_propnames:
                propvalue = npc.attributes.get(propname, default="N/A")
                output += "\n %s = %s" % (propname, propvalue)
            caller.msg(output)
        elif self.propname not in allowed_propnames:
            caller.msg("You may only change %s." %
                               ", ".join(allowed_propnames))
        elif self.propval:
            # assigning a new propvalue
            # in this example, the properties are all integers...
            intpropval = int(self.propval)
            npc.attributes.add(self.propname, intpropval)
            caller.msg("Set %s's property '%s' to %s" %
                        (npc.key, self.propname, self.propval))
        else:
            # propname set, but not propval - show current value
            caller.msg("%s has property %s = %s" %
                        (npc.key, self.propname,
                         npc.attributes.get(self.propname, default="N/A")))
```

This command example shows off the use of more advanced parsing but otherwise it's mostly error checking. It searches for the given npc in the same room, and checks so the caller actually has permission to "edit" it before continuing. A player without the proper permission won't even be able to view the properties on the given NPC. It's

up to each game if this is the way it should be.

Add this to the default command set like before and you should be able to try it out.

Note: If you wanted a player to use this command to change an on-object property like the NPC's name (the ''key'' property), you'd need to modify the command since "key" is not an Attribute (it is not retrievable via ''npc.attributes.get'' but directly via ''npc.key''). We leave this as an optional exercise.

Making the NPC do stuff - the +npc command

Finally, we will make a command to order our NPC around. For now, we will limit this command to only be usable by those having the "edit" permission on the NPC. This can be changed if it's possible for anyone to use the NPC.

The NPC, since it inherited our Character typeclass has access to most commands a player does. What it doesn't have access to are Session and Player-based cmdsets (which means, among other things that they cannot chat on channels, but they could do that if you just added those commands). This makes the `+npc` command simple:

```
+npc Anna = say Hello!
Anna says, 'Hello!'
```

Again, add to the end of your `command.py` module:

```
class CmdNPC(Command):
    """
    controls an NPC

    Usage:
        +npc <name> = <command>

    This causes the npc to perform a command as itself. It will do so
    with its own permissions and accesses.
    """
    key = "+npc"
    locks = "call:not perm(nonpcs)"
    help_category = "mush"

    def parse(self):
        "Simple split of the = sign"
        name, cmdname = None, None
        if "=" in self.args:
            name, cmdname = [part.strip()
                             for part in self.args.rsplit("=", 1)]
        self.name, self.cmdname = name, cmdname

    def func(self):
        "Run the command"
        caller = self.caller
        if not self.cmdname:
            caller.msg("Usage: +npc <name> = <command>")
            return
        npc = caller.search(self.name)
        if not npc:
            return
        if not npc.access(caller, "edit"):
            caller.msg("You may not order this NPC to do anything.")
            return
        # send the command order
```

```
        npc.execute_cmd(self.cmdname)
        caller.msg("You told %s to do '%s'." % (npc.key, self.cmdname))
```

Note that if you give an erroneous command, you will not see any error message, since that error will be returned to the npc object, not to you. If you want players to see this, you can give the caller's session ID to the `execute_cmd` call, like this:

```
npc.execute_cmd(self.cmdname, sessid=self.caller.sessid)
```

Another thing to remember is however that this is a very simplistic way to control NPCs. Evennia supports full puppeting very easily. A Player (assuming the "puppet" permission was set correctly) could simply do `@ic mynpc` and be able to play the game "as" that NPC. This is in fact just what happens when a Player takes control of their normal Character as well.

Concluding remarks

This ends the tutorial. It looks like a lot of text but the amount of code you have to write is actually relatively short. At this point you should have a basic skeleton of a game and a feel for what is involved in coding your game.

From here on you could build a few more ChargenRooms and link that to a bigger grid. The `+setpower` command can either be built upon or accompanied by many more to get a more elaborate character generation.

The simple "Power" game mechanic should be easily expandable to something more full-fledged and useful, same is true for the combat score principle. The `+attack` could be made to target a specific player (or npc) and automatically compare their relevant attributes to determine a result.

To continue from here, you can take a look at the Tutorial World. For more specific ideas, see the other tutorials and hints as well
as the Developer Central.

Adding Command Tutorial

This is a quick first-time tutorial expanding on the Commands documentation.

Let's assume you have just downloaded Evennia, installed it and created your game folder (let's call it just `mygame` here). Now you want to try to add a new command. This is the fastest way to do it.

Step 1: Creating a custom command

1. Open `mygame/commands/command.py` in a text editor. It already contains some example code.
2. Create a new class in `command.py` inheriting from `default_cmds.MuxCommand`. Let's call it `CmdEcho` in this example.
3. Set the class variable `key` to a good command name, like `echo`.
4. Give your class a useful `__doc__` string, this will become the help entry for the Command (see Command Auto-help).
5. Define a class method `func()` that does stuff.

Below is an example how this all could look:

```
# file mygame/commands/command.py
#[...]
from evennia import default_cmds
class CmdEcho(default_cmds.MuxCommand):
    """
    Simple command example

    Usage:
      echo [text]

    This command simply echoes text back to the caller.
    """

    key = "echo"

    def func(self):
        "This actually does things"
        if not self.args:
            self.caller.msg("You didn't enter anything!")
        else:
            self.caller.msg("You gave the string: '%s'" % self.args)
```

Step 2: Adding the Command to a default Cmdset

The command is not available to use until it is part of a Command Set. In this example we will go the easiest route and add it to the default Character commandset that already exists.

1. Edit `mygame/commands/default_cmdsets.py`
2. Import your new command with `from commands.command import CmdEcho`.
3. Add a line `self.add(CmdEcho())` to `CharacterCmdSet`, in the `at_cmdset_creation` method (the template tells you where).

This is approximately how it should look at this point:

```
# file mygame/commands/default_cmdsets.py
#[...]
from commands.command import CmdEcho
#[...]
class CharacterCmdSet(default_cmds.CharacterCmdSet):

    key = "DefaultCharacter"

    def at_cmdset_creation(self):

        # this first adds all default commands
        super(DefaultSet, self).at_cmdset_creation()

        # all commands added after this point will extend or
        # overwrite the default commands.
        self.add(CmdEcho())
```

Next, run the `@reload` command. You should now be able to use your new `echo` command from inside the game. Use `help echo` to see the documentation for the command.

If you have trouble, make sure to check the log for error messages (probably due to syntax errors in your command definition).

If you want to overload existing default commands (such as `look` or `get`), just add your new command with the same key as the old one - it will then replace it. Just remember that you must use `@reload` to see any changes.

See Commands for many more details and possibilities when defining Commands and using Cmdsets in various ways.

Adding the command to specific object types

Adding your Command to the `CharacterCmdSet` is just one easy exapmple. The cmdset system is very generic. You can create your own cmdsets (let's say in a module `mycmdsets.py`) and add them to objects as you please (how to control their merging is described in detail in the Command Set documentation).

```
# file mygame/commands/mycmdsets.py
#[...]
from commands.command import CmdEcho
from evennia import CmdSet
#[...]
class MyCmdSet(CmdSet):

    key = MyCmdSet

    def at_cmdset_creation(self):
        self.add(CmdEcho())
```

Now you just need to add this to an object. To test things (as superuser) you can do

```
@py self.cmdset.add("mycmdsets.MyCmdSet")
```

This will add this cmdset (along with its echo command) to yourself so you can test it. Note that you cannot add a single Command to an object on its own, it must be part of a CommandSet in order to do so.

The Command you added is not there permanently at this point. If you do a `@reload` the merger will be gone. You *could* add the `permanent=True` keyword to the `cmdset.add` call. This will however only make the new merged cmdset permanent on that *single* object. Often you want *all* objects of this particular class to have this cmdset.

To make sure all new created objects get your new merged set, put the `cmdset.add` call in your custom Typeclasses' `at_object_creation` method:

```
# e.g. in mygame/typeclasses/objects.py

from evennia import DefaultObject
class MyObject(DefaultObject):

    def at_object_creation(self):
        "called when the object is first created"
        self.cmdset.add("mycmdset.MyCmdSet", permanent=True)
```

All new objects of this typeclass will now start with this cmdset and it will survive a `@reload`.

Note: An important caveat with this is that `at_object_creation` is only called *once*, when the object is first created. This means that if you already have existing objects in your databases using that typeclass, they will not have been initiated the same way. There are many ways to update them; since it's a one-time update you can usually just simply loop through them. As superuser, try the following:

```
@py from typeclasses.objects import MyObject; [o.cmdset.add("mycmdset.MyCmdSet") for␣
↪o in MyObject.objects.all()]
```

This goes through all objects in your database having the right typeclass, adding the new cmdset to each. The good news is that you only have to do this if you want to post-add *cmdsets*. If you just want to add a new *command*,

you can simply add that command to the cmdset's `at_cmdset_creation` and `@reload` to make the Command immediately available.

Change where Evennia looks for command sets

Evennia uses settings variables to know where to look for its default command sets. These are normally not changed unless you want to re-organize your game folder in some way. For example, the default character cmdset defaults to being defined as

```
CMDSET_CHARACTER="commands.default_cmdset.CharacterCmdSet"
```

See `evennia/settings_default` for the other settings.

Adding Object Typeclass Tutorial

Evennia comes with a few very basic classes of in-game entities:

```
DefaultObject
   |
   DefaultCharacter
   DefaultRoom
   DefaultExit
```

When you create a new Evennia game (with for example `evennia --init mygame`) Evennia will automatically create empty child classes `Object`, `Character`, `Room` and `Exit` respectively. They are found `mygame/typeclasses/objects.py`, `mygame/typeclasses/rooms.py` etc.

> Technically these are all Typeclassed, which can be ignored for now. In `mygame/typeclasses` are also base typeclasses for out-of-character things, notably Channels, Players and Scripts. We don't cover those in this tutorial.

For your own game you will most likely want to expand on these very simple beginnings. It's normal to want your Characters to have various attributes, for example. Maybe Rooms should hold extra information or even *all* Objects in your game should have properties not included in basic Evennia.

Change Default Rooms, Exits, Character Typeclass

This is the simplest case.

The default build commands of a new Evennia game is set up to use the `Room`, `Exit` and `Character` classes found in the same-named modules under `mygame/typeclasses/`. By default these are empty and just implements the default parents from the Evennia library (`DefaultRoom`etc). Just add the changes you want to these classes and run `@reload` to add your new functionality.

Create a new type of object

Say you want to create a new "Heavy" object-type that characters should not have the ability to pick up.

1. Edit `mygame/typeclasses/objects.py` (you could also create a new module there, named something like `heavy.py`, that's up to how you want to organize things).
2. Create a new class inheriting at any distance from `DefaultObject`. It could look something like this:

```
# end of file mygame/typeclasses/objects.py
from evennia import DefaultObject

class Heavy(DefaultObject):
   "Heavy object"
   def at_object_creation(self):
       "Called whenever a new object is created"
       # lock the object down by default
       self.locks.add("get:false()")
       # the default "get" command looks for this Attribute in order
       # to return a customized error message (we just happen to know
       # this, you'd have to look at the code of the 'get' command to
       # find out).
       self.db.get_err_msg = "This is too heavy to pick up."
```

3. Once you are done, log into the game with a build-capable account and do `@create/drop rock:objects.Heavy` to drop a new heavy "rock" object in your location. Next try to pick it up (`@quell` yourself first if you are a superuser). If you get errors, look at your log files where you will find the traceback. The most common error is that you have some sort of syntax error in your class.

Note that the Locks and Attribute which are set in the typeclass could just as well have been set using commands in-game, so this is a *very* simple example.

Storing data on initialization

The `at_object_creation` is only called once, when the object is first created. This makes it ideal for database-bound things like Attributes. But sometimes you want to create temporary properties (things that are not to be stored in the database but still always exist every time the object is created). Such properties can be initialized in the `at_init` method on the object. `at_init` is called every time the object is loaded into memory.

> Note: It's usually pointless and wasteful to assign database data in `at_init`, since this will hit the database with the same value over and over. Put those in `at_object_creation` instead.

You are wise to use `ndb` (non-database Attributes) to store non-persistent these properties, since ndb-properties are protected against being cached out in various ways and also allows you to list them using various in-game tools:

```
def at_init(self):
    self.ndb.counter = 0
    self.ndb.mylist = []

Note: As mentioned in the `Typeclasses`_ documentation, ``at_init``
replaces the use of the standard ``__init__`` method of typeclasses
due to how the latter may be called in situations other than you'd
expect. So use ``at_init`` where you would normally use
``__init__``.
```

Updating existing objects

If you already have some `Heavy` objects created and you add a new `Attribute` in `at_object_creation`, you will find that those existing objects will not have this Attribute. This is not so strange, since `at_object_creation` is only called once, it will not be called again just because you update it. You need to update existing objects manually.

If the number of objects is limited, you can use `@typeclass/force/reload objectname` to force a re-load of the `at_object_creation` method (only) on the object. This case is common enough that there is an alias

`@update objectname` you can use to get the same effect. If there are multiple objects you can use `@py` to loop over the objects you need:

```
@py from typeclasses.objects import Heavy; [obj.at_object_creation() for obj in Heavy.
↪objects.all()]
```

Tutorial World Introduction

The *Tutorial World* is a small and functioning MUD-style game world. It is intended to be deconstructed and used as a way to learn Evennia. The game consists of a single-player quest and has some 20 rooms that you can explore as you seek to discover the whereabouts of a mythical weapon.

The source code is fully documented. You can find the whole thing in `evennia/contrib/tutorial_world/`.

Some features exemplified by the tutorial world:

- Tutorial command, giving "behind-the-scenes" help for every room and some of the special objects
- Rooms with custom `return_appearance` to show details.
- Hidden exits
- Objects with multiple custom interactions
- Large-area rooms
- Outdoor weather rooms
- Dark room, needing light source
- Puzzle object
- Multi-room puzzle
- Aggressive mobile with roam, pursue and battle state-engine AI
- Weapons, also used by mobs
- Simple combat system with attack/defend commands
- Object spawning
- Teleporter trap rooms

Install

The tutorial world consists of a few modules in `evennia/contrib/tutorial_world/` containing custom Typeclasses for rooms and objects and associated Commands.

These reusable bits and pieces are then put together into a functioning game area ("world" is maybe too big a word for such a small zone) using a batch script called `build.ev`. To install, log into the server as the superuser (user #1) and run:

```
@batchcommand tutorial_world.build
```

The world will be built (this might take a while, so don't rerun the command even if it seems the system has frozen). After finishing you will end up back in Limbo with a new exit called `tutorial`.

An alternative is

```
@batchcommand/interactive tutorial_world.build
```

with the /interactive switch you are able to step through the building process at your own pace to see what happens in detail.

To play the tutorial "correctly", you should *not* do so as superuser. The reason for this is that many game systems ignore the presence of a superuser and will thus not work as normal. Use the `@quell` command to limit your powers or log out and reconnect as a different user. As superuser you can of course examine things "under the hood" later if you want.

Gameplay

To get into the mood of this miniature quest, imagine you are an adventurer out to find fame and fortune. You have heard rumours of an old castle ruin by the coast. In its depth a warrior princess was buried together with her powerful magical weapon - a valuable prize, if it's true. Of course this is a chance to adventure that you cannot turn down!

You reach the ocean in the midst of a raging thunderstorm. With wind and rain screaming in your face you stand where the moor meets the sea along a high, rocky coast ...

- Look at everything.
- Some objects are interactive in more than one way. Use the normal `help` command to get a feel for which commands are available at any given time. (use the command `tutorial` to get insight behind the scenes of the tutorial).
- In order to fight, you need to first find some type of weapon.
- *slash* is a normal attack
- *stab* launches an attack that makes more damage but has a lower chance to hit.
- *defend* will lower the chance to taking damage on your enemy's next attack.
- You *can* run from a fight that feels too deadly. Expect to be chased though.
- Being defeated is a part of the experience ...

Uninstall

Uninstalling the tutorial world basically means deleting all the rooms and objects it consists of. First, move out of the tutorial area.

```
@find tut#01
@find tut#17
```

This should locate the first and last rooms created by `build.ev` - *Intro* and *Outro*. If you installed normally, everything created between these two numbers should be part of the tutorial. Note their dbref numbers, for example 5 and 80. Next we just delete all objects in that range:

```
@del 5-80
```

You will see some errors since some objects are auto-deleted and so cannot be found when the delete mechanism gets to them. That's fine. You should have removed the tutorial completely once the command finishes.

Notes

When reading and learning from the code, keep in mind that *Tutorial World* was created with a very specific goal: to install easily and to not permanently modify the rest of the server. It therefore goes to some length to use only temporary solutions and to clean up after
itself.

Command Duration

Note: This is an advanced tutorial, don't try it until you well understand how 'Commands'_ work.

In some types of games a command should not start and finish immediately. Loading a crossbow might take a bit of time to do - time you don't have when the enemy comes rushing at you. Crafting that armour will not be immediate either. For some types of games the very act of moving or changing pose all comes with a certain time associated with it.

Below is a simple command example for adding a duration for a command to finish.

```
from evennia import default_cmds, utils

class CmdEcho(default_cmds.MuxCommand):
    """
    wait for an echo

    Usage:
      echo <string>

    Calls and waits for an echo
    """
    key = "echo"
    locks = "cmd:all()"

    def func(self):
        """
         This is called at the initial shout.
        """
        self.caller.msg("You shout '%s' and wait for an echo ..." % self.args)
        # this waits non-blocking for 10 seconds, then calls self.echo
        utils.delay(10, callback=self.echo) # call echo after 10 seconds

    def echo(self):
        "Called after 10 seconds."
        shout = self.args
        string = "You hear an echo: %s ... %s ... %s"
        string = string % (shout.upper(), shout.capitalize(), shout.lower())
        self.caller.msg(string)
```

Import this new echo command into the default command set and reload the server. You will find that it will take 10 seconds before you see your shout coming back. You will also find that this is a *non-blocking* effect; you can issue other commands in the interim and the game will go on as usual. The echo will come back to you in its own time.

About utils.delay()

`utils.delay(delay, callback=None, *args, **kwargs)` is a useful function. It will wait `delay` seconds, then call a function you give it as `callback(*args, **kwargs)`.

If you are not familiar with the syntax `*args` and `**kwargs`, see the Python documentation here.

Looking at it you might think that `utils.delay(10, callback)` in the code above is just an alternative to some more familiar thing like `time.sleep(10)`. This is *not* the case. If you do `time.sleep(10)` you will in fact freeze the *entire server* for ten seconds! The `utils.delay()` is a thin wrapper around a Twisted Deferred that will delay execution until 10 seconds have passed, but will do so asynchronously, without bothering anyone else (not even you - you can continue to do stuff normally while it waits to continue).

The point to remember here is that the `delay()` call will not "pause" at that point when it is called. The lines after the `delay()` call will actually execute right away. What you must do is to tell it which function to call *after the time has passed* (its "callback"). This may sound strange at first, but it is normal practice in asynchronous systems. You can also link such calls together as seen below:

```
from evennia import default_cmds, utils

class CmdEcho(default_cmds.MuxCommand):
    """
    waits for an echo

    Usage:
      echo <string>

    Calls and waits for an echo
    """
    key = "echo"
    locks = "cmd:all()"

    def func(self):
        "This sets off a chain of delayed calls"

        self.caller.msg("You shout '%s', waiting for an echo ..." % self.args)

        # wait 2 seconds before calling self.echo1
        utils.delay(2, callback=self.echo1)

    # callback chain, started above
    def echo1(self):
        "First echo"
        self.caller.msg("... %s" % self.args.upper())
        # wait 2 seconds for the next one
        utils.delay(2, callback=self.echo2)

    def echo2(self):
        "Second echo"
        self.caller.msg("... %s" % self.args.capitalize())
        # wait another 2 seconds
        utils.delay(2, callback=self.echo3)

    def echo3(self):
        "Last echo"
        self.caller.msg("... %s ..." % self.args.lower())
```

The above version will have the echoes arrive one after another, each separated by a two second delay.

```
> echo Hello!
... HELLO!
... Hello!
... hello! ...
```

Blocking commands

As mentioned, a great thing about the delay introduced by `utils.delay()` is that it does not block. It just goes on in the background and you are free to play normally in the interim. In some cases this is not what you want however. Some commands should simply "block" other commands while they are running. If you are in the process of crafting a helmet you shouldn't be able to also start crafting a shield at the same time, or if you just did a huge power-swing with your weapon you should not be able to do it again immediately.

The simplest way of implementing blocking is to use the technique covered in the Command Cooldown tutorial. In that tutorial we implemented cooldowns by having the Command store the current time. Next time the Command was called, we compared the current time to the stored time to determine if enough time had passed for a renewed use. This is a *very* efficient, reliable and passive solution. The drawback is that there is nothing to tell the Player when enough time has passed unless they keep trying.

Here is an example where we will use `utils.delay` to tell the player when the cooldown has passed:

```
from evennia import utils, default_cmds

class CmdBigSwing(default_cmds.MuxCommand):
    """
    swing your weapon in a big way

    Usage:
      swing <target>

    Makes a mighty swing. Doing so will make you vulnerable
    to counter-attacks before you can recover.
    """
    key = "bigswing"
    locks = "cmd:all()"

    def func(self):
        "Makes the swing"

        if self.caller.ndb.off_balance:
            # we are still off-balance.
            self.caller.msg("You are off balance and need time to recover!")
            return

        # [attack/hit code goes here ...]

        self.caller.msg("You swing big! You are off balance now.")

        # set the off-balance flag
        self.caller.ndb.off_balance = True

        # wait 8 seconds before we can recover. During this time
        # we won't be able to swing again due to the check at the top.
        utils.delay(8, callback=self.recover)

    def recover(self):
        "This will be called after 8 secs"
        del self.caller.ndb.off_balance
        self.caller.msg("You regain your balance.")
```

Note how, after the cooldown, the user will get a message telling them they are now ready for another swing.

By storing the `off_balance` flag on the character (rather than on, say, the Command instance itself) it can be accessed by other Commands too. Other attacks may also not work when you are off balance. You could also have an

enemy Command check your `off_balance` status to gain bonuses, to take another example.

Abortable commands

One can imagine that you will want to abort a long-running command before it has a time to finish. If you are in the middle of crafting your armor you will probably want to stop doing that when a monster enters your smithy.

You can implement this in the same way as you do the "blocking" command above, just in reverse. Below is an example of a crafting command that can be aborted by starting a fight:

```
from evennia import utils, default_cmds

class CmdCraftArmour(default_cmds.MuxCommand):
    """
    Craft armour

    Usage:
       craft <name of armour>

    This will craft a suit of armour, assuming you
    have all the components and tools. Doing some
    other action (such as attacking someone) will
    abort the crafting process.
    """
    key = "craft"
    locks = "cmd:all()"

    def func(self):
        "starts crafting"

        if self.caller.ndb.is_crafting:
            self.caller.msg("You are already crafting!")
            return
        if self._is_fighting():
            self.caller.msg("You can't start to craft "
                            "in the middle of a fight!")
            return

        # [Crafting code, checking of components, skills etc]

        # Start crafting
        self.caller.ndb.is_crafting = True
        self.caller.msg("You start crafting ...")
        utils.delay(60, callback=self.step1)

    def _is_fighting(self):
        "checks if we are in a fight."
        if self.caller.ndb.is_fighting:
            del self.caller.ndb.is_crafting
            return True

    def step1(self):
        "first step of armour construction"
        if self._is_fighting():
            return
        self.msg("You create the first part of the armour.")
        utils.delay(60, callback=self.step2)
    def step2(self):
```

```
        "second step of armour construction"
        if self._is_fighting():
            return
        self.msg("You create the second part of the armour.")
        utils.delay(60, callback=step3)
    def step3(self):
        "last step of armour construction"
        if self._is_fighting():
            return

        # [code for creating the armour object etc]

        del self.caller.ndb.is_crafting
        self.msg("You finalize your armour.")


# example of a command that aborts crafting

class CmdAttack(default_cmds.MuxCommand):
    """
    attack someone

    Usage:
        attack <target>

    Try to cause harm to someone. This will abort
    eventual crafting you may be currently doing.
    """
    key = "attack"
    aliases = ["hit", "stab"]
    locks = "cmd:all()"

    def func(self):
        "Implements the command"

        self.caller.ndb.is_fighting = True

        # [...]
```

The above code creates a delayed crafting command that will gradually create the armour. If the `attack` command is issued during this process it will set a flag that causes the crafting to be quietly canceled next time it tries to update.

Assorted Notes

In these examples we only used `utils.delay()`, which is a very simple wrapper around Twisted's `reactor.callLater()`. If you know your Twisted one might imagine using more advanced features such as callback/errback chains to more efficiently handle various command states and conditions.

Command Prompt

A *prompt* is quite common in MUDs. The prompt display useful details about your character that you are likely to want to keep tabs on at all times, such as health, magical power etc. It might also show things like in-game time, weather and so on. Many modern MUD clients (including Evennia's own webclient) allows for identifying the prompt

and have it appear in a correct location (usually just above the input line). Usually it will remain like that until it is explicitly updated.

Sending a prompt

A prompt is sent using the `prompt` keyword to the `msg()` method on objects. The prompt will be sent without any line breaks.

```
self.msg(prompt="HP: 5, MP: 2, SP: 8")
```

You can combine the sending of normal text with the sending (updating of the prompt):

```
self.msg("This is a text", prompt="This is a prompt")
```

You can update the prompt on demand, this is normally done using OOB-tracking of the relevant Attributes (like the character's health). You could also make sure that attacking commands update the prompt when they cause a change in health, for example.

Here is a simple example of the prompt sent/updated from a command class:

```
from evennia import Command

class CmdDiagnose(Command):
    """
    see how hurt your are

    Usage:
      diagnose [target]

    This will give an estimate of the target's health. Also
    the target's prompt will be updated.
    """
    key = "diagnose"

    def func(self):
        if not self.args:
            target = self.caller
        else:
            target = self.search(self.args)
            if not target:
                return
        # try to get health, mana and stamina
        hp = target.db.hp
        mp = target.db.mp
        sp = target.db.sp

        if None in (hp, mp, sp):
            # Attributes not defined
            self.caller.msg("Not a valid target!")
            return

        text = "You diagnose %s as having " \
               "%i health, %i mana and %i stamina." \
               % (hp, mp, sp)
        prompt = "%i HP, %i MP, %i SP" % (hp, mp, sp)
        self.caller.msg(text, prompt=prompt)
```

A prompt sent with every command

The prompt sent as described above uses a standard telnet instruction (the Evennia web client gets a special flag). Most MUD telnet clients will understand and allow users to catch this and keep the prompt in place until it updates. So *in principle* you'd not need to update the prompt every command.

However, with a varying user base it can be unclear which clients are used and which skill level the users have. So sending a prompt with every command is a safe catch-all. You don't need to manually go in and edit every command you have though. Instead you edit the base command class for your custom commands (like `MuxCommand` in your `mygame/commands/command.py` folder) and overload the `at_post_cmd()` hook. This hook is always called *after* the main `func()` method of the Command.

```
from evennia import default_cmds

class MuxCommand(default_cmds.MuxCommand):
    # ...
    def at_post_cmd(self):
        "called after self.func()."
        caller = self.caller
        prompt = "%i HP, %i MP, %i SP" % (caller.db.hp,
                                          caller.db.mp,
                                          caller.db.sp)
        caller.msg(prompt=prompt)
```

Modifying default commands

If you want to add something small like this to Evennia's default commands without modifying them directly the easiest way is to just wrap those with a multiple inheritance to your own base class:

```
# in (for example) mygame/commands/mycommands.py

from evennia import default_cmds
# our custom MuxCommand with at_post_cmd hook
from commands.command import MuxCommand

# overloading the look command
class CmdLook(default_cmds.CmdLook, MuxCommand):
    pass
```

The result of this is that the hooks from your custom `MuxCommand` will be mixed into the default `CmdLook` through multiple inheritance. Next you just add this to your default command set:

```
# in mygame/commands/default_cmdsets.py

from evennia import default_cmds
from commands import mycommands

class CharacterCmdSet(default_cmds.CharacterCmdSet):
    # ...
    def at_cmdset_creation(self):
        # ...
        self.add(mycommands.CmdLook())
```

This will automatically replace the default `look` command in your game with your own version.

Default Exit Errors

Evennia allows for exits to have any name. The command "kitchen" is a valid exit name as well as "jump out the window" or "north". An exit actually consists of two parts: an Exit Object and an Exit Command stored on said exit object. The command has the same key and aliases as the object, which is why you can see the exit in the room and just write its name to traverse it.

If you try to enter the name of a non-existing exit, it is thus the same as trying a non-exising command; Evennia doesn't care about the difference:

```
> jump out the window
Command 'jump out the window' is not available. Type "help" for help.
```

Many games don't need this type of freedom however. They define only the cardinal directions as valid exit names (Evennia's `@tunnel` command also offers this functionality). In this case, the error starts to look less logical:

```
> west
Command 'west' is not available. Maybe you meant "@set" or "@reset"?
```

Since we for our particular game *know* that west is an exit direction, it would be better if the error message just told us that we couldn't go there.

Adding default error commands

To solve this you need to be aware of how to write and add new commands. What you need to do is to create new commands for all directions you want to support in your game. In this example all we'll do is echo an error message, but you could certainly consider more advanced uses. You add these commands to the default command set. Here is an example of such a set of commands:

```
# for example in a file mygame/commands/movecommands.py

from evennia import default_cmds

class CmdExitError(default_cmds.MuxCommand):
    "Parent class for all exit-errors."
    locks = "cmd:all()"
    arg_regex = r"\s|$"
    auto_help = False
    def func(self):
        "returns the error"
        self.caller.msg("You cannot move %s." % self.key)

class CmdExitErrorNorth(CmdExitError):
    key = "north"
    aliases = ["n"]

class CmdExitErrorEast(CmdExitError):
    key = "east"
    aliases = ["e"]

class CmdExitErrorSouth(CmdExitError):
    key = "south"
    aliases = ["s"]

class CmdExitErrorWest(CmdExitError):
```

```
    key = "west"
    aliases = ["w"]
```

Make sure to add the directional commands (not their parent) to the `CharacterCmdSet` class in `mygame/commands/default_cmdsets.py`:

```
# in mygame/commands/default_cmdsets.py

from commands import movecommands

# [...]
class CharacterCmdSet(default_cmds.CharacterCmdSet):
    # [...]
    def at_cmdset_creation(self):
        # [...]
        self.add(movecommands.CmdExitErrorNorth())
        self.add(movecommands.CmdExitErrorEast())
        self.add(movecommands.CmdExitErrorSouth())
        self.add(movecommands.CmdExitErrorWest())
```

After a `@reload` these commands (assuming you don't get any errors - check your log) will be loaded. What happens henceforth is that if you are in a room with an Exitobject (let's say it's "north"), the proper Exit-command will overload your error command (also named "north"). But if you enter an direction without having a matching exit for it, you will fallback to your default error commands:

```
> east
You cannot move east.
```

Further expansions by the exit system (including manipulating the way the Exit command itself is created) can be done by modifying the Exit typeclass directly.

Additional Comments

So why didn't we create a single error command above? Something like this:

```
class CmdExitError(default_cmds.MuxCommand):
   "Handles all exit-errors."
   key = "error_cmd"
   aliases = ["north", "n",
              "east", "e",
              "south", "s",
              "west", "w"]
    #[...]
```

The anwer is that this would *not* work and understanding why is important in order to not be confused when working with commands and command sets.

The reason it doesn't work is because Evennia's command system compares commands *both* by `key` and by `aliases`. If *either* of those match, the two commands are considered *identical* as far as cmdset merging system is concerned.

So the above example would work fine as long as there were no Exits at all in the room. But what happens when we enter a room with an exit "north"? The Exit's cmdset is merged onto the default one, and since there is an alias match, the system determines our `CmdExitError` to be identical. It is thus overloaded by the Exit command (which also correctly defaults to a higher priority). The result is that you can go through the north exit normally but none of the

error messages for the other directions are available since the single error command was completely overloaded by the single matching "north" exit-command.

Implementing a game rule system

The simplest way to create an online roleplaying game (at least from a code perspective) is to simply grab a paperback RPG rule book, get a staff of game masters together and start to run scenes with whomever logs in. Game masters can roll their dice in front of their computers and tell the players the results. This is only one step away from a traditional tabletop game and puts heavy demands on the staff - it is unlikely staff will be able to keep up around the clock even if they are very dedicated.

Many games, even the most roleplay-dedicated, thus tend to allow for players to mediate themselves to some extent. A common way to do this is to introduce *coded systems* - that is, to let the computer do some of the heavy lifting. A basic thing is to add an online dice-roller so everyone can make rolls and make sure noone is cheating. Somewhere at this level you find the most bare-bones roleplaying MUSHes.

The advantage of a coded system is that as long as the rules are fair the computer is too - it makes no judgement calls and holds no personal grudges (and cannot be accused of holding any). Also, the computer doesn't need to sleep and can always be online regardless of when a player logs on. The drawback is that a coded system is not flexible and won't adapt to the unprogrammed actions human players may come up with in role play. For this reason many roleplay-heavy MUDs do a hybrid variation - they use coded systems for things like combat and skill progression but leave role play to be mostly freeform, overseen by staff game masters.

Finally, on the other end of the scale are less- or no-roleplay games, where game mechanics (and thus player fairness) is the most important aspect. In such games the only events with in-game value are those resulting from code. Such games are very common and include everything from hack-and-slash MUDs to various tactical simulations.

So your first decision needs to be just what type of system you are aiming for. This page will try to give some ideas for how to organize the "coded" part of your system, however big that may be.

Overall system infrastructure

We strongly recommend that you code your rule system as stand-alone as possible. That is, don't spread your skill check code, race bonus calculation, die modifiers or what have you all over your game.

- Put everything you would need to look up in a rule book into a module in `mygame/world`. Hide away as much as you can. Think of it as a black box (or maybe the code representation of an all-knowing game master). The rest of your game will ask this black box questions and get answers back. Exactly how it arrives at those results should not need to be known outside the box. Doing it this way makes it easier to change and update things in one place later.
- Store only the minimum stuff you need with each game object. That is, if your Characters need values for Health, a list of skills etc, store those things on the Character - don't store how to roll or change them.
- Next is to determine just how you want to store things on your Objects and Characters. You can choose to either store things as individual Attributes, like `character.db.STR=34` and `character.db.Hunting_skill=20`. But you could also use some custom storage method, like a dictionary `character.db.skills = {"Hunting":34, "Fishing":20, ...}`. Finally you could even go with a custom django model. Which is the better depends on your game and the complexity of your system.
- Make a clear API into your rules. That is, make methods/functions that you feed with, say, your Character and which skill you want to check. That is, you want something similar to this:

```
from world import rules
result = rules.roll_skill(character, "hunting")
result = rules.roll_challenge(character1, character2, "swords")
```

You might need to make these functions more or less complex depending on your game. For example the properties of the room might matter to the outcome of a roll (if the room is dark, burning etc). Establishing just what you need to send into your game mechanic module is a great way to also get a feel for what you need to add to your engine.

Coded systems

Inspired by tabletop role playing games, most game systems mimic some sort of die mechanic. To this end Evennia offers a full dice roller in its `contrib` folder. For custom implementations, Python offers many ways to randomize a result using its in-built `random` module. No matter how it's implemented, we will in this text refer to the action of determining an outcome as a "roll".

In a freeform system, the result of the roll is just compared with values and people (or the game master) just agree on what it means. In a coded system the result now needs to be processed somehow. There are many things that may happen as a result of rule enforcement:

- Health may be added or deducted. This can effect the character in various ways.
- Experience may need to be added, and if a level-based system is used, the player might need to be informed they have increased a level.
- Room-wide effects need to be reported to the room, possibly affecting everyone in the room.

There are also a slew of other things that fall under "Coded systems", including things like weather, NPC artificial intelligence and game economy. Basically everything about the world that a Game master would control in a tabletop role playing game can be mimicked to some level by coded systems.

Example of Rule module

Here is a simple example of a rule module. This is what we assume about our simple example game:

- Characters have only four numerical values:
- Their `level`, which starts at 1.
- A skill `combat`, which determines how good they are at hitting things. Starts between 5 and 10.
- Their Strength, `STR`, which determine how much damage they do. Starts between 1 and 10.
- Their Health points, `HP`, which starts at 100.
- When a Character reaches `HP = 0`, they are presumed "defeated". Their HP is reset and they get a failure message (as a stand-in for death code).
- Abilities are stored as simple Attributes on the Character.
- "Rolls" are done by rolling a 100-sided die. If the result is below the `combat` value, it's a success and damage is rolled. Damage is rolled as a six-sided die + the value of `STR` (for this example we ignore weapons and assume `STR` is all that matters).
- Every successful `attack` roll gives 1-3 experience points (`XP`). Every time the number of `XP` reaches `(level + 1) ** 2`, the Character levels up. When leveling up, the Character's `combat` value goes up by 2 points and `STR` by one (this is a stand-in for a real progression system).

Character

The Character typeclass is simple. It goes in `mygame/typeclasses/characters.py`. There is already an empty `Character` class there that Evennia will look to and use.

```
from random import randint
from evennia import DefaultCharacter

class Character(DefaultCharacter):
    """
    Custom rule-restricted character. We randomize
    the initial skill and ability values bettween 1-10.
    """
    def at_object_creation(self):
        "Called only when first created"
        self.db.level = 1
        self.db.HP = 100
        self.db.XP = 0
        self.db.STR = randint(1, 10)
        self.db.combat = randint(5, 10)
```

`@reload` the server to load up the new code. Doing `examine self` will however *not* show the new Attributes on yourself. This is because the `at_object_creation` hook is only called on *new* Characters. Your Character was already created and will thus not have them. To force a reload, use the following command:

```
@typeclass/force/reset self
```

The `examine self` command will now show the new Attributes.

Rule module

This is a module `mygame/world/rules.py`.

```
from random import randint

def roll_hit():
    "Roll 1d100"
    return randint(1, 100)

def roll_dmg():
    "Roll 1d6"
    return randint(1, 6)

def check_defeat(character):
    "Checks if a character is 'defeated'."
    if character.db.HP <= 0:
       character.msg("You fall down, defeated!")
       character.db.HP = 100   # reset

def add_XP(character, amount):
    "Add XP to character, tracking level increases."
    character.db.XP += amount
    if character.db.XP >= (character.db.level + 1) ** 2:
        character.db.level += 1
        character.db.STR += 1
        character.db.combat += 2
        character.msg("You are now level %i!" % character.db.level)

def skill_combat(*args):
    """
    This determines outcome of combat. The one who
    rolls under their combat skill AND higher than
```

```
    their opponent's roll hits.
    """
    char1, char2 = args
    roll1, roll2 = roll_hit(), roll_hit()
    failtext = "You are hit by %s for %i damage!"
    wintext = "You hit %s for %i damage!"
    xp_gain = randint(1, 3)
    if char1.db.combat >= roll1 > roll2:
        # char 1 hits
        dmg = roll_dmg() + char1.db.STR
        char1.msg(wintext % (char2, dmg))
        add_XP(char1, xp_gain)
        char2.msg(failtext % (char1, dmg))
        char2.db.HP -= dmg
        check_defeat(char2)
    elif char2.db.combat >= roll2 > roll1:
        # char 2 hits
        dmg = roll_dmg() + char2.db.STR
        char1.msg(failtext % (char2, dmg))
        char1.db.HP -= dmg
        check_defeat(char1)
        char2.msg(wintext % (char1, dmg))
        add_XP(char2, xp_gain)
    else:
        # a draw
        drawtext = "Neither of you can find an opening."
        char1.msg(drawtext)
        char2.msg(drawtext)

SKILLS = {"combat": skill_combat}

def roll_challenge(character1, character2, skillname):
    """
    Determine the outcome of a skill challenge between
    two characters based on the skillname given.
    """
    if skillname in SKILLS:
        SKILLS[skillname](character1,-character2)
    else:
        raise RunTimeError("Skillname %s not found." % skillname)
```

These few functions implement the entirety of our simple rule system. We have a function to check the "defeat" condition and reset the `HP` back to 100 again. We define a generic "skill" function. Multiple skills could all be added with the same signature; our `SKILLS` dictionary makes it easy to look up the skills regardless of what their actual functions are called. Finally, the access function `roll_challenge` just picks the skill and gets the result.

In this example, the skill function actually does a lot - it not only rolls results, it also informs everyone of their results via `character.msg()` calls.

Here is an example of usage in a game command:

```
from evennia import Command
from world import rules

class CmdAttack(Command):
    """
    attack an opponent
```

```
    Usage:
      attack <target>

    This will attack a target in the same room, dealing
    damage with your bare hands.
    """
    def func(self):
        "Implementing combat"

        caller = self.caller
        if not self.args:
            caller.msg("You need to pick a target to attack.")
            return

        target = caller.search(self.args)
        if target:
            rules.roll_challenge(caller, target, "combat")
```

Note how simple the command becomes and how generic you can make it. It becomes simple to offer any number of Combat commands by just extending this functionality - you can easily roll challenges and pick different skills to check. And if you ever decided to, say, change how to determine hit chance, you don't have to change every command, but need only change the single `roll_hit` function inside your `rules` module.

Weather Tutorial

This tutorial will have us create a simple weather system for our MUD. The way we want to use this is to have all outdoor rooms echo weather-related messages to the room at regular and semi-random intervals. Things like "Clouds gather above", "It starts to rain" and so on.

One could imagine every outdoor room in the game having a script running on themselves that fires regularly. For this particular example it is however more efficient to do it another way, namely by using a "ticker-subscription" model. The principle is simple: Instead of having each Object individually track the time, they instead subscribe to be called by a global ticker who handles time keeping. Not only does this centralize and organize much of the code in one place, it also has less computing overhead.

Evennia offers the TickerHandler specifically for using the subscription model. We will use it for our weather system.

We will assume you know how to make your own Typeclasses. If not see one of the beginning tutorials. We will create a new WeatherRoom typeclass that is aware of the day-night cycle.

```
import random
from evennia import DefaultRoom, TICKER_HANDLER

ECHOES = ["The sky is clear.",
          "Clouds gather overhead.",
          "It's starting to drizzle.",
          "A breeze of wind is felt.",
          "The wind is picking up"] # etc

class WeatherRoom(DefaultRoom):
    "This room is ticked at regular intervals"

    def at_object_creation(self):
        "called only when the object is first created"
        TICKER_HANDLER.add(60 * 60, self.at_weather_update)
```

```
    def at_weather_update(self, *args, **kwargs):
        "ticked at regular intervals"
        echo = random.choice(ECHOES)
        self.msg_contents(echo)
```

In the `at_object_creation` method, we simply added ourselves to the TickerHandler and tell it to call `at_weather_update` every hour (`60*60` seconds). During testing you might want to play with a shorter time duration.

For this to work we also create a custom hook `at_weather_update(*args, **kwargs)`, which is the call sign required by TickerHandler hooks.

Henceforth the room will inform everyone inside it when the weather changes. This particular example is of course very simplistic - the weather echoes are just randomly chosen and don't care what weather came before it. Expanding it to be more realistic is a useful exercise.

Zones

Say you create a room named *Meadow* in your nice big forest MUD. That's all nice and dandy, but what if you, in the other end of that forest want another *Meadow*? As a game creator, this can cause all sorts of confusion. For example, teleporting to *Meadow* will now give you a warning that there are two *Meadow* s and you have to select which one. It's no problem to do that, you just choose for example to go to `2-meadow`, but unless you examine them you couldn't be sure which of the two sat in the magical part of the forest and which didn't.

Another issue is if you want to group rooms in geographic regions. Let's say the "normal" part of the forest should have separate weather patterns from the magical part. Or maybe a magical disturbance echoes through all magical-forest rooms. It would then be convenient to be able to simply find all rooms that are "magical" so you could send messages to them.

Zones in Evennia

Zones try to separate rooms by global location. In our example we would separate the forest into two parts - the magical and the non-magical part. Each have a *Meadow* and rooms belonging to each part should be easy to retrieve.

Many MUD codebases hardcode zones as part of the engine and database. Evennia does no such distinction due to the fact that rooms themselves are meant to be customized to any level anyway. Below is a suggestion for how to implement zones in Evennia.

All objects in Evennia can hold any number of Tags. Tags are short labels that you attach to objects. They make it very easy to retrieve groups of objects. An object can have any number of different tags. So let's attach the relevant tag to our forest:

```
forestobj.tags.add("magicalforest", category="zone")
```

You could add this manually, or automatically during creation somehow (you'd need to modify your @dig command for this, most likely). You can also use the default `@tag` command during building:

```
@tag forestobj = magicalforest : zone
```

Henceforth you can then easily retrieve only objects with a given tag:

```
import evennia
rooms = evennia.search_tag("magicalforest", category="zone")
```

Using typeclasses and inheritance for zoning

The tagging or aliasing systems above don't instill any sort of functional difference between a magical forest room and a normal one - they are just arbitrary ways to mark objects for quick retrieval later. Any functional differences must be expressed using Typeclasses.

Of course, an alternative way to implement zones themselves is to have all rooms/objects in a zone inherit from a given typeclass parent - and then limit your searches to objects inheriting from that given parent. The effect would be similar but you'd need to expand the search functionality to properly search the inheritance tree.

Web Tutorial

Evennia uses the Django web framework as the basis of both its database configuration and the website it provides. While a full understanding of Django requires reading the Django documentation, we have provided this tutorial to get you running with the basics and how they pertain to Evennia. This text details getting everything set up. The Web-based Character view Tutorial gives a more explicit example of making a custom web page connected to your game, and you may want to read that after finishing this guide.

A Basic Overview

Django is a web framework. It gives you a set of development tools for building a website quickly and easily.

Django projects are split up into *apps* and these apps all contribute to one project. For instance, you might have an app for conducting polls, or an app for showing news posts or, like us, one for creating a web client.

Each of these applications has a `urls.py` file, which specifies what URLs are used by the app, a `views.py` file for the code that the URLs activate, a `templates` directory for displaying the results of that code in HTML for the user, and a `static` folder that holds assets like CSS, Javascript, and Image files (You may note your mygame/web folder does not have a `static` or `template` folder. This is intended and explained further below). Django applications may also have a `models.py` file for storing information in the database. We will not change any models here, take a look at the New Models page (as well as the Django docs on models) if you are interested.

There is also a root `urls.py` that determines the URL structure for the entire project. A starter `urls.py` is included in the default game template, and automatically imports all of Evennia's default URLs for you. This is located in `web/urls.py`.

Changing the logo on the front page

Evennia's default logo is a fun little googly-eyed snake wrapped around a gear globe. As cute as it is, it probably doesn't represent your game. So one of the first things you may wish to do is replace it with a logo of your own.

Django web apps all have *static assets*: CSS files, Javascript files, and Image files. In order to make sure the final project has all the static files it needs, the system collects the files from every app's `static` folder and places it in the `STATIC_ROOT` defined in `settings.py`. By default, the Evennia `STATIC_ROOT` is in `web/static`.

Because Django pulls files from all of those separate places and puts them in one folder, it's possible for one file to overwrite another. We will use this to plug in our own files without having to change anything in the Evennia itself.

By default, Evennia is configured to pull files you put in the `web/static_overrides` *after* all other static files. That means that files in `static_overrides` folder will overwrite any previously loaded files *having the same path under its static folder*. This last part is important to repeat: To overload the static resource from a standard `static` folder you need to replicate the path of folders and file names from that `static` folder in exactly the same way inside `static_overrides`.

Let's see how this works for our logo. The default web application is in the Evennia library itself, in `evennia/web/`. We can see that there is a `static` folder here. If we browse down, we'll eventually find the full path to the Evennia logo file: `evennia/web/static/evennia_general/images/evennia_logo.png`.

Inside our `static_overrides` we must replicate the part of the path inside the `static` folder, in other words, we must replicate `evennia_general/images/evennia_logo.png`.

So, to change the logo, we need to create the folder path `evennia_general/images/` in `static_overrides`. We then rename our own logo file to `evennia_logo.png` and copy it there. The final path for this file would thus be: `web/static_overrides/evennia_general/images/evennia_logo.png` in your local game folder.

To get this file pulled in, just change to your own game directory and reload the server:

```
evennia reload
```

This will reload the configuration and bring in the new static file(s). If you didn't want to reload the server you could instead use

```
evennia collectstatic
```

to only update the static files without any other changes.

> **Note**: Evennia will collect static files automatically during startup. So if `evennia collectstatic` reports finding 0 files to collect, make sure you didn't start the engine at some point - if so the collector has already done its work! To make sure, connect to the website and check so the logo has actually changed to your own version.
>
> **Note**: Sometimes the static asset collector can get confused. If no matter what you do, your overridden files aren't getting copied over the defaults, try removing the target file (or everything) in the `web/static` directory, and re-running `collectstatic` to gather everything from scratch.

Changing the Front Page's Text

The default front page for Evennia contains information about the Evennia project. You'll probably want to replace this information with information about your own project. Changing the page template is done in a similar way to changing static resources.

Like static files, Django looks through a series of template folders to find the file it wants. The difference is that Django does not copy all of the template files into one place, it just searches through the template folders until it finds a template that matches what it's looking for. This means that when you edit a template, the changes are instant. You don't have to reload the server or run any extra commands to see these changes - reloading the web page in your browser is enough.

To replace the index page's text, we'll need to find the template for it. We'll go into more detail about how to determine which template is used for rendering a page in the Web-based Character view Tutorial. For now, you should know that the template we want to change is stored in `evennia/web/website/templates/website/index.html`.

To replace this template file, you will put your changed template inside the `web/template_overrides/website` directory in your game folder. In the same way as with static resources you must replicate the path inside the default `template` directory exactly. So we must copy our replacement template named `index.html` there (or create the `website` directory in web/template_overrides`if it does not exist, first). The final path to the file should thus be:` web/template_overrides/website/index.html‘ within your game directory.

Note that it is usually easier to just copy the original template over and edit it in place. The original file already has all the markup and tags, ready for editing.

Further reading

For further hints on working with the web presence, you could now continue to the Web-based Character view Tutorial where you learn to make a web page that displays in-game character stats. You can also look at Django's own tutorial to get more insight in how Django works and what possibilities exist.

Web Character View Tutorial

Before doing this tutorial you will probably want to read the intro in 'Basic Web tutorial'_.

In this tutorial we will create a web page that displays the stats of a game character. For this, and all other pages we want to make specific to our game, we'll need to create our own Django "app"

We'll call our app `character`, since it will be dealing with character information. From your game dir, run

```
evennia startapp character
```

This will create a directory named `character` in the root of your game dir. It contains all basic files that a Django app needs. To keep `mygame` well ordered, move it to your `mygame/web/` directory instead:

```
mv character web/
```

Note that we will not edit all files in this new directory, many of the generated files are outside the scope of this tutorial.

In order for Django to find our new web app, we'll need to add it to the `INSTALLED_APPS` setting. Evennia's default installed apps are already set, so in `server/conf/settings.py`, we'll just extend them:

```
INSTALLED_APPS += ('web.character',)

Note: That end comma is important. It makes sure that Python
interprets the addition as a tuple instead of a string.
```

The first thing we need to do is to create a *view* and an *URL pattern* to point to it. A view is a function that generates the web page that a visitor wants to see, while the URL pattern lets Django know what URL should trigger the view. The pattern may also provide some information of its own as we shall see.

Here is our `character/urls.py` file (**Note**: you may have to create this file if a blank one wasn't generated for you):

```
# URL patterns for the character app

from django.conf.urls import url
from web.character.views import sheet

urlpatterns = [
    url(r'^sheet/(?P<object_id>\d+)/$', sheet, name="sheet")
]
```

This file contains all of the URL patterns for the application. The `url` function in the `urlpatterns` list are given three arguments. The first argument is a pattern-string used to identify which URLs are valid. Patterns are specified as *regular expressions*. Regular expressions are used to match strings and are written in a special, very compact, syntax. A detailed description of regular expressions is beyond this tutorial but you can learn more about them here. For now, just accept that this regular expression requires that the visitor's URL looks something like this:

```
sheet/123/
```

That is, `sheet/` followed by a number, rather than some other possible URL pattern. We will interpret this number as object ID. Thanks to how the regular expression is formulated, the pattern recognizer stores the number in a variable called `object_id`. This will be passed to the view (see below). We add the imported view function (`sheet`) in the second argument. We also add the `name` keyword to identify the URL pattern itself. You should always name your URL patterns, this makes them easy to refer to in html templates using the `{% url %}` tag (but we won't get more into that in this tutorial).

> Security Note: Normally, users do not have the ability to see object IDs within the game (it's restricted to superusers only). Exposing the game's object IDs to the public like this enables griefers to perform what is known as an account enumeration attack in the efforts of hijacking your superuser account. Consider this: in every Evennia installation, there are two objects that we can *always* expect to exist and have the same object IDs– Limbo (#2) and the superuser you create in the beginning (#1). Thus, the griefer can get 50% of the information they need to hijack the admin account (the admin's username) just by navigating to `sheet/1`!

Next we create `views.py`, the view file that `urls.py` refers to.

```
# Views for our character app

from django.http import Http404
from django.shortcuts import render
from django.conf import settings

from evennia.utils.search import object_search
from evennia.utils.utils import inherits_from

def sheet(request, object_id):
    object_id = '#' + object_id
    try:
        character = object_search(object_id)[0]
    except IndexError:
        raise Http404("I couldn't find a character with that ID.")
    if not inherits_from(character, settings.BASE_CHARACTER_TYPECLASS):
        raise Http404("I couldn't find a character with that ID. "
                      "Found something else instead.")
    return render(request, 'character/sheet.html', {'character': character})
```

As explained earlier, the URL pattern parser in `urls.py` parses the URL and passes `object_id` to our view function `sheet`. We do a database search for the object using this number. We also make sure such an object exists and that it is actually a Character. The view function is also handed a `request` object. This gives us information about the request, such as if a logged-in user viewed it - we won't use that information here but it is good to keep in mind.

On the last line, we call the `render` function. Apart from the `request` object, the `render` function takes a path to an html template and a dictionary with extra data you want to pass into said template. As extra data we pass the Character object we just found. In the template it will be available as the variable "character".

The html template is created as `templates/character/sheet.html` under your `character` app folder. You may have to manually create both `template` and its subfolder `character`. Here's the template to create:

```
{% extends "base.html" %}
{% block content %}

    <h1>{{ character.name }}</h1>

    <p>{{ character.db.desc }}</p>

    <h2>Stats</h2>
```

```
    <table>
      <thead>
        <tr>
          <th>Stat</th>
          <th>Value</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Strength</td>
          <td>{{ character.db.str }}</td>
        </tr>
        <tr>
          <td>Intelligence</td>
          <td>{{ character.db.int }}</td>
        </tr>
        <tr>
          <td>Speed</td>
          <td>{{ character.db.spd }}</td>
        </tr>
      </tbody>
    </table>

    <h2>Skills</h2>
    <ul>
      {% for skill in character.db.skills %}
        <li>{{ skill }}</li>
      {% empty %}
        <li>This character has no skills yet.</li>
      {% endfor %}
    </ul>

    {% if character.db.approved %}
      <p class="success">This character has been approved!</p>
    {% else %}
      <p class="warning">This character has not yet been approved!</p>
    {% endif %}
{% endblock %}
```

In Django templates, `{% ... %}` denotes special in-template "functions" that Django understands. The `{{ ... }}` blocks work as "slots". They are replaced with whatever value the code inside the block returns.

The first line, `{% extends "base.html" %}`, tells Django that this template extends the base template that Evennia is using. The base template is provided by the theme. Evennia comes with the open-source third-party theme `prosimii`. You can find it and its `base.html` in `evennia/web/templates/prosimii`. Like other templates, these can be overwritten.

The next line is `{% block content %}`. The `base.html` file has `block`s, which are placeholders that templates can extend. The main block, and the one we use, is named `content`.

We can access the `character` variable anywhere in the template because we passed it in the `render` call at the end of `view.py`. That means we also have access to the Character's `db` attributes, much like you would in normal Python code. You don't have the ability to call functions with arguments in the template– in fact, if you need to do any complicated logic, you should do it in `view.py` and pass the results as more variables to the template. But you still have a great deal of flexibility in how you display the data.

We can do a little bit of logic here as well. We use the `{% for %} ... {% endfor %}` and `{% if %} ... {% else %} ... {% endif %}` structures to change how the template renders depending on how many skills

the user has, or if the user is approved (assuming your game has an approval system).

The last file we need to edit is the master URLs file. This is needed in order to smoothly integrate the URLs from your new `character` app with the URLs from Evennia's existing pages. Find the file `web/urls.py` and update its `patterns` list as follows:

```
# web/urls.py

custom_patterns = [
    url(r'^character/', include('web.character.urls',
                                namespace='character', app_name='character')),
    ]
```

Now reload the server with `evennia reload` and visit the page in your browser. If you haven't changed your defaults, you should be able to find the sheet for character `#1` at `http://localhost:8000/character/sheet/1/`

Try updating the stats in-game and refresh the page in your browser. The results should show immediately.

As an optional final step, you can also change your character typeclass to have a method called 'get_absolute_url'.

```
# typeclasses/characters.py

    # inside Character
    def get_absolute_url(self):
        from django.core.urlresolvers import reverse
        return reverse('character:sheet', kwargs={'object_id':self.id})
```

Doing so will give you a 'view on site' button in the top right of the Django Admin Objects changepage that links to your new character sheet, and allow you to get the link to a character's page by using {{ object.get_absolute_url }} in any template where you have a given object.

Now that you've made a basic page and app with Django, you may want to read the full Django tutorial to get a better idea of what it can do. 'You can find Django's tutorial here'_.

CHAPTER 14

Appendix

This chapter groups various extra resources and pieces of information.

Links

A list of resources that may be useful for Evennia users and developers.

Evennia links

- evennia.com - Main Evennia portal page. Links to all corners of Evennia.
- Evennia github page - Download code and read documentation.
- Evennia forums/mailing list - Web interface to our google group.
- Evennia development blog - Musings from the lead developer.
- Evennia's manual on ReadTheDocs - Read and download offline in html, PDF or epub formats.
- **'Evennia Game Index'_** - An automated listing of Evennia games.

Evennia game projects

If your game is online, please consider adding it to the 'Evennia game index'_ instead!

- The world of Cool battles - Open to try out! A fun combat-arena oriented Evennia game, code also available on github.
- ArxMUSH - A comprehensive MUSH done in Evennia, in beta-testing
- Nalvia - Open to try out! Telnet to `game.nalvia.net`, port `8000` (no webclient at this time)
- Ainneve - The Evennia example game (under development) (forum).
- nextRPI - A github project for making a toolbox for people to make RPI-style Evennia games.

- Muddery - A mud framework under development, based on Evennia. It has some specific design goals for building and extending the game based on input files. ‘‘_ ‘ <–%5BLatitude%5D(https://github.com/dbenoy/latitude)%20(MUCK%20under%20development,%20using%20Evennia)>‘__
- *‘Language Understanding for Text games using Deep reinforcement learning‘*_ (PDF) - MIT research paper using Evennia to train AIs.
- Evennia-related repos on github

Third-party Evennia utilities and resources

Let us know if you don’t find your project here!

- EvCast video series - Tutorial videos explaining installing Evennia, basic Python etc.
- Evennia-docker - Evennia in a Docker container for quick install and deployment in just a few commands.
- EvenniaWinPE - An independently maintained ‘portable’ version of Evennia. It is created semi-regularly from the latest sources and ships with all dependencies pre-installed in a zip file (Windows only). *Note that this is currently over a year out-of-date*.
- Evennia’s docs in Chinese - A translated mirror of a slightly older Evennia version. Announcement [here] (https://groups.google.com/forum/#!topic/evennia/3AXS8ZTzJaA).
- Evennia for MUSHers - An article describing Evennia for those used to the MUSH way of doing things.
- vim-evennia - A mode for editing batch-build files (`.ev`) files in the vim text editor (Emacs users can use evennia-mode.el).
- Evennia on Open Hub
- Evennia on OpenHatch
- Evennia on PyPi
- notimetoplay post about Evennia
- Evennia subreddit (not much there yet though)

Other useful mud development resources

- ROM area reader - Parser for converting ROM area files to Python objects.

General MUD forums and discussions

- Imaginary Realities - An e-magazine on game and MUD design that has several articles about Evennia. There is also an archive of older issues from 1998-2001 that are still very relevant.
- MuSoapbox - Very active Mu* game community mainly focused on MUSH-type gaming.
- Optional Realities - Mud development discussion forums that has regular articles on MUD development focused on roleplay-intensive games. After a HD crash it’s not as content-rich as it once was.
- MudLab - Mud design discussion forum
- MudConnector - Mud listing and forums
- MudBytes - Mud listing and forums
- Top Mud Sites - Mud listing and forums

- Planet Mud-Dev - A blog aggregator following blogs of current MUD development (including Evennia) around the 'net. Worth to put among your RSS subscriptions.
- Mud Dev mailing list archive (mirror) - Influential mailing list active 1996-2004. Advanced game design discussions.
- Mud-dev wiki - A (very) slowly growing resource on MUD creation.
- Mud Client/Server Interaction - A page on classic MUD telnet protocols.
- Mud Tech's fun/cool but ... - Greg Taylor gives good advice on mud design.
- Lost Library of MOO - Archive of scientific articles on mudding (in particular moo).
- Nick Gammon's hints thread - Contains a very useful list of things to think about when starting your new MUD.
- Lost Garden - A game development blog with long and interesting articles (not MUD-specific)
- What Games Are - A blog about general game design (not MUD-specific)
- The Alexandrian - A blog about tabletop roleplaying and board games, but with lots of general discussion about rule systems and game balance that could be applicable also for MUDs.

Literature

- Richard Bartle *Designing Virtual Worlds* (amazon page) - Essential reading for the design of any persistent game world, written by the co-creator of the original game *MUD*. Published in 2003 but it's still as relevant now as when it came out. Covers everything you need to know and then some.
- Zed A. Shaw *Learn Python the Hard way* (homepage) - Despite the imposing name this book is for the absolute Python/programming beginner. One learns the language by gradually creating a small text game! It has been used by multiple users before moving on to Evennia. All chapters can be read for free from the homepage. (Ignore the author's stance on Python3 though, that is just bizarre).
- David M. Beazley *Python Essential Reference (4th ed)* (amazon page) - Our recommended book on Python; it not only efficiently summarizes the language but is also an excellent reference to the standard library for more experienced Python coders.
- Luciano Ramalho, *Fluent Python* (o'reilly page) - This is an excellent book for experienced Python coders willing to take their code to the next level. A great read with a lot of useful info also for veteran Pythonistas.
- Richard Cantillon *An Essay on Economic Theory* (free pdf) - A very good English translation of *Essai sur la Nature du Commerce en Général*, one of the foundations of modern economic theory. Written in 1730 but the translation is annotated and the essay is actually very easy to follow also for a modern reader. Required reading if you think of implementing a sane game economic system.

Frameworks

- Django's homepage
- Documentation
- Code
- Twisted homepage
- Documentation
- Code

Tools

- GIT
- Documentation
- Learn GIT in 15 minutes (interactive tutorial)

Python Info

- Python Website
- Documentation
- Tutorial
- Library Reference
- Language Reference
- Python tips and tricks

Credits

- Wiki Home Icons made by Freepik from flaticon.com, licensed under Creative Commons BY 3.0.

Default Command Help

This wiki page is auto-generated. Do not modify, your changes will be lost.

The full set of default Evennia commands currently contains 89 commands in 9 source files.
Our policy for adding default commands is outlined here. More information about how
commands work can be found in the documentation for Commands.

A-Z

- @about - show Evennia info
- @alias - adding permanent aliases for object
- @ban - ban a player from the server
- @batchcode - build from batch-code file
- @batchcommands - build from batch-command file
- @boot - kick a player from the server.
- @cboot - kick a player from a channel you control
- @ccreate - create a new channel
- @cdesc - describe a channel you control
- @cdestroy - destroy a channel you created

- @cemit - send an admin message to a channel you control
- @channels - list all channels available to you
- @charcreate - create a new character
- @clock - change channel locks of a channel you control
- @cmdsets - list command sets defined on an object
- @color - testing which colors your client support
- @copy - copy an object and its properties
- @cpattr - copy attributes between objects
- @create - create new objects
- @cwho - show who is listening to a channel
- @delplayer - delete a player from the server
- @desc - describe an object
- @destroy - permanently delete objects
- @dig - build new rooms and connect them to the current location
- @emit - admin command for emitting message to multiple objects
- @examine - get detailed information about an object
- @find - search the database for objects
- @help - Edit the help database.
- @home - set an object's home location
- @ic - control an object you have permission to puppet
- @irc2chan - link an evennia channel to an external IRC channel
- @link - link existing rooms together with exits
- @lock - assign a lock definition to an object
- @mvattr - move attributes between objects
- @name - change the name and/or aliases of an object
- @objects - statistics on objects in the database
- @ooc - stop puppeting and go ooc
- @open - open a new exit from the current room
- @option - Set an account option
- @password - change your password
- @perm - set the permissions of a player/object
- @py - execute a snippet of python code
- @quell - use character's permissions instead of player's
- @quit - quit the game
- @reload - reload the server
- @reset - reset and reboot the server

- @rss2chan - link an evennia channel to an external RSS feed
- @script - attach a script to an object
- @scripts - list and manage all running scripts
- @server - show server load and memory statistics
- @service - manage system services
- @sessions - check your connected session(s)
- @set - set attribute on an object or player
- @shutdown - stop the server completely
- @spawn - spawn objects from prototype
- @tag - handles the tags of an object
- @tel - teleport object to another location
- @time - show server time statistics
- @tunnel - create new rooms in cardinal directions only
- @typeclass - set or change an object's typeclass
- @unban - remove a ban from a player
- @unlink - remove exit-connections between rooms
- @userpassword - change the password of a player
- @wall - make an announcement to all
- @wipe - clear all attributes from an object
- __unloggedin_look_command - look when in unlogged-in state
- access - show your current game access
- addcom - add a channel alias and/or subscribe to a channel
- allcom - perform admin operations on all channels
- command - This is a parent class for some of the defining objmanip commands
- connect - connect to the game
- create - create a new player account
- delcom - remove a channel alias and/or unsubscribe from channel
- drop - drop something
- get - pick up something
- give - give away something to someone
- help - get help when in unconnected-in state
- help - View help or a list of topics
- home - move to your character's home location
- inventory - view inventory
- look - look at location or object
- look - look while out-of-character

- nick - define a personal alias/nick
- page - send a private message to another player
- pose - strike a pose
- quit - quit when in unlogged-in state
- say - speak as your character
- whisper - Speak privately as your character to another
- who - list who is currently online

Command details

These are generated from the auto-documentation and are ordered by their source file location in `evennia/commands/default/`.

admin.py

View Source

@ban (CmdBan)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@ban“
- aliases = [‘@bans‘]
- locks = “cmd:perm(ban) or perm(Immortals)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
ban a player from the server

Usage:
  @ban [<name or ip> [: reason]]

Without any arguments, shows numbered list of active bans.

This command bans a user from accessing the game. Supply an optional
reason to be able to later remember why the ban was put in place.

It is often preferable to ban a player from the server than to
delete a player with @delplayer. If banned by name, that player
account can no longer be logged into.

IP (Internet Protocol) address banning allows blocking all access
from a specific address or subnet. Use an asterisk (*) as a
wildcard.

Examples:
  @ban thomas             - ban account 'thomas'
  @ban/ip 134.233.2.111   - ban specific ip address
```

```
  @ban/ip 134.233.2.*      - ban all in a subnet
  @ban/ip 134.233.*.*      - even wider ban

A single IP filter can be easy to circumvent by changing computers
or requesting a new IP address. Setting a wide IP block filter with
wildcards might be tempting, but remember that it may also
accidentally block innocent users connecting from the same country
or region.
```

open source (admin.py)

@boot (CmdBoot)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@boot“
- aliases = []
- locks = “cmd:perm(boot) or perm(Wizards)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
kick a player from the server.

Usage
  @boot[/switches] <player obj> [: reason]

Switches:
  quiet - Silently boot without informing player
  sid - boot by session id instead of name or dbref

Boot a player object from the server. If a reason is
supplied it will be echoed to the user unless /quiet is set.
```

open source (admin.py)

@delplayer (CmdDelPlayer)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@delplayer“
- aliases = []
- locks = “cmd:perm(delplayer) or perm(Immortals)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
delete a player from the server

Usage:
  @delplayer[/switch] <name> [: reason]

Switch:
```

```
  delobj - also delete the player's currently
           assigned in-game object.

Completely deletes a user from the server database,
making their nick and e-mail again available.
```

open source (admin.py)

@emit (CmdEmit)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@emit“
- aliases = [‘@remit‘, ‘@pemit‘]
- locks = “cmd:perm(emit) or perm(Builders)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
admin command for emitting message to multiple objects

Usage:
  @emit[/switches] [<obj>, <obj>, ... =] <message>
  @remit           [<obj>, <obj>, ... =] <message>
  @pemit           [<obj>, <obj>, ... =] <message>

Switches:
  room : limit emits to rooms only (default)
  players : limit emits to players only
  contents : send to the contents of matched objects too

Emits a message to the selected objects or to
your immediate surroundings. If the object is a room,
send to its contents. @remit and @pemit are just
limited forms of @emit, for sending to rooms and
to players respectively.
```

open source (admin.py)

@perm (CmdPerm)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@perm“
- aliases = [‘@setperm‘]
- locks = “cmd:perm(perm) or perm(Immortals)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
set the permissions of a player/object

Usage:
```

```
  @perm[/switch] <object> [= <permission>[,<permission>,...]]
  @perm[/switch] *<player> [= <permission>[,<permission>,...]]

Switches:
  del : delete the given permission from <object> or <player>.
  player : set permission on a player (same as adding * to name)

This command sets/clears individual permission strings on an object
or player. If no permission is given, list all permissions on <object>.
```

open source (admin.py)

@unban (CmdUnban)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@unban“
- aliases = []
- locks = “cmd:perm(unban) or perm(Immortals)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
remove a ban from a player

Usage:
  @unban <banid>

This will clear a player name/ip ban previously set with the @ban
command.  Use this command without an argument to view a numbered
list of bans. Use the numbers in this list to select which one to
unban.
```

open source (admin.py)

@userpassword (CmdNewPassword)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@userpassword“
- aliases = []
- locks = “cmd:perm(newpassword) or perm(Wizards)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
change the password of a player

Usage:
  @userpassword <user obj> = <new password>

Set a player's password.
```

open source (admin.py)

@wall (CmdWall)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@wall“
- aliases = []
- locks = “cmd:perm(wall) or perm(Wizards)”
- help_category = “Admin”
- __doc__ string (auto-help):

```
make an announcement to all

Usage:
  @wall <message>

Announces a message to all connected players.
```

open source (admin.py)

batchprocess.py

View Source

@batchcode (CmdBatchCode)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@batchcode“
- aliases = [‘@batchcodes‘]
- locks = “cmd:superuser()”
- help_category = “Building”
- __doc__ string (auto-help):

```
build from batch-code file

Usage:
 @batchcode[/interactive] <python path to file>

Switch:
   interactive - this mode will offer more control when
                 executing the batch file, like stepping,
                 skipping, reloading etc.
   debug - auto-delete all objects that has been marked as
           deletable in the script file (see example files for
           syntax). This is useful so as to to not leave multiple
           object copies behind when testing out the script.

Runs batches of commands from a batch-code text file (*.py).
```

open source (batchprocess.py)

@batchcommands (CmdBatchCommands)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@batchcommands“
- aliases = [‘@batchcmd‘, ‘@batchcommand‘]
- locks = “cmd:perm(batchcommands) or superuser()”
- help_category = “Building”
- __doc__ string (auto-help):

```
build from batch-command file

Usage:
 @batchcommands[/interactive] <python.path.to.file>

Switch:
   interactive - this mode will offer more control when
                 executing the batch file, like stepping,
                 skipping, reloading etc.

Runs batches of commands from a batch-cmd text file (*.ev).
```

open source (batchprocess.py)

building.py

View Source

@alias (CmdSetObjAlias)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@alias“
- aliases = [‘@setobjalias‘]
- locks = “cmd:perm(setobjalias) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
adding permanent aliases for object

Usage:
  @alias <obj> [= [alias[,alias,alias,...]]]
  @alias <obj> =

Assigns aliases to an object so it can be referenced by more
than one name. Assign empty to remove all aliases from object.

Observe that this is not the same thing as personal aliases
```

```
created with the 'nick' command! Aliases set with @alias are
changing the object in question, making those aliases usable
by everyone.
```

open source (building.py)

@cmdsets (CmdListCmdSets)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@cmdsets“
- aliases = [‘@listcmsets‘]
- locks = “cmd:perm(listcmdsets) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
list command sets defined on an object

Usage:
  @cmdsets [obj]

This displays all cmdsets assigned
to a user. Defaults to yourself.
```

open source (building.py)

@copy (CmdCopy)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@copy“
- aliases = []
- locks = “cmd:perm(copy) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
copy an object and its properties

Usage:
  @copy[/reset] <original obj> [= new_name][;alias;alias..][:new_location] [,new_
↪name2 ...]

switch:
  reset - make a 'clean' copy off the object, thus
          removing any changes that might have been made to the original
          since it was first created.

Create one or more copies of an object. If you don't supply any targets,
one exact copy of the original object will be created with the name *_copy.
```

open source (building.py)

@cpattr (CmdCpAttr)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@cpattr“
- aliases = []
- locks = “cmd:perm(cpattr) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
copy attributes between objects

Usage:
  @cpattr[/switch] <obj>/<attr> = <obj1>/<attr1> [,<obj2>/<attr2>,<obj3>/<attr3>,...]
  @cpattr[/switch] <obj>/<attr> = <obj1> [,<obj2>,<obj3>,...]
  @cpattr[/switch] <attr> = <obj1>/<attr1> [,<obj2>/<attr2>,<obj3>/<attr3>,...]
  @cpattr[/switch] <attr> = <obj1>[,<obj2>,<obj3>,...]

Switches:
  move - delete the attribute from the source object after copying.

Example:
  @cpattr coolness = Anna/chillout, Anna/nicety, Tom/nicety
  ->
  copies the coolness attribute (defined on yourself), to attributes
  on Anna and Tom.

Copy the attribute one object to one or more attributes on another object.
If you don't supply a source object, yourself is used.
```

open source (building.py)

@create (CmdCreate)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@create“
- aliases = []
- locks = “cmd:perm(create) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
create new objects

Usage:
  @create[/drop] objname[;alias;alias...][:typeclass], objname...

switch:
   drop - automatically drop the new object into your current
          location (this is not echoed). This also sets the new
          object's home to the current location rather than to you.
```

```
Creates one or more new objects. If typeclass is given, the object
is created as a child of this typeclass. The typeclass script is
assumed to be located under types/ and any further
directory structure is given in Python notation. So if you have a
correct typeclass 'RedButton' defined in
types/examples/red_button.py, you could create a new
object of this type like this:

   @create/drop button;red : examples.red_button.RedButton
```

open source (building.py)

@desc (CmdDesc)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@desc“
- aliases = [‘@describe‘]
- locks = “cmd:perm(desc) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
describe an object

Usage:
  @desc [<obj> =] <description>

Switches:
  edit - Open up a line editor for more advanced editing.

Sets the "desc" attribute on an object. If an object is not given,
describe the current room.
```

open source (building.py)

@destroy (CmdDestroy)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@destroy“
- aliases = [‘@del‘, ‘@delete‘]
- locks = “cmd:perm(destroy) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
permanently delete objects

Usage:
   @destroy[/switches] [obj, obj2, obj3, [dbref-dbref], ...]

switches:
```

```
   override - The @destroy command will usually avoid accidentally
              destroying player objects. This switch overrides this safety.
examples:
   @destroy house, roof, door, 44-78
   @destroy 5-10, flower, 45

Destroys one or many objects. If dbrefs are used, a range to delete can be
given, e.g. 4-10. Also the end points will be deleted.
```

open source (building.py)

@dig (CmdDig)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@dig“
- aliases = []
- locks = “cmd:perm(dig) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
build new rooms and connect them to the current location

Usage:
  @dig[/switches] roomname[;alias;alias...][:typeclass]
        [= exit_to_there[;alias][:typeclass]]
           [, exit_to_here[;alias][:typeclass]]

Switches:
   tel or teleport - move yourself to the new room

Examples:
   @dig kitchen = north;n, south;s
   @dig house:myrooms.MyHouseTypeclass
   @dig sheer cliff;cliff;sheer = climb up, climb down

This command is a convenient way to build rooms quickly; it creates the
new room and you can optionally set up exits back and forth between your
current room and the new one. You can add as many aliases as you
like to the name of the room and the exits in question; an example
would be 'north;no;n'.
```

open source (building.py)

@examine (CmdExamine)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@examine“
- aliases = [‘examine’, ‘@ex‘, ‘ex’, ‘exam’]
- locks = “cmd:perm(examine) or perm(Builders)”

- help_category = "Building"
- __doc__ string (auto-help):

```
get detailed information about an object

Usage:
  examine [<object>[/attrname]]
  examine [*<player>[/attrname]]

Switch:
  player - examine a Player (same as adding *)
  object - examine an Object (useful when OOC)

The examine command shows detailed game info about an
object and optionally a specific attribute on it.
If object is not specified, the current location is examined.

Append a * before the search string to examine a player.
```

open source (building.py)

@find (CmdFind)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@find"
- aliases = ['locate', '@locate', 'search', '@search', 'find']
- locks = "cmd:perm(find) or perm(Builders)"
- help_category = "Building"
- __doc__ string (auto-help):

```
search the database for objects

Usage:
  @find[/switches] <name or dbref or *player> [= dbrefmin[-dbrefmax]]

Switches:
  room - only look for rooms (location=None)
  exit - only look for exits (destination!=None)
  char - only look for characters (BASE_CHARACTER_TYPECLASS)
  exact- only exact matches are returned.

Searches the database for an object of a particular name or exact #dbref.
Use *playername to search for a player. The switches allows for
limiting object matches to certain game entities. Dbrefmin and dbrefmax
limits matches to within the given dbrefs range, or above/below if only
one is given.
```

open source (building.py)

@home (CmdSetHome)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@home“
- aliases = [‘@sethome‘]
- locks = “cmd:perm(@home) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
set an object's home location

Usage:
  @home <obj> [= home_location]

The "home" location is a "safety" location for objects; they
will be moved there if their current location ceases to exist. All
objects should always have a home location for this reason.
It is also a convenient target of the "home" command.

If no location is given, just view the object's home location.
```

open source (building.py)

@link (CmdLink)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@link“
- aliases = []
- locks = “cmd:perm(link) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
link existing rooms together with exits

Usage:
  @link[/switches] <object> = <target>
  @link[/switches] <object> =
  @link[/switches] <object>

Switch:
  twoway - connect two exits. For this to work, BOTH <object>
           and <target> must be exit objects.

If <object> is an exit, set its destination to <target>. Two-way operation
instead sets the destination to the *locations* of the respective given
arguments.
The second form (a lone =) sets the destination to None (same as
the @unlink command) and the third form (without =) just shows the
currently set destination.
```

open source (building.py)

@lock (CmdLock)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@lock“
- aliases = [‘lock’, ‘@locks‘, ‘locks’]
- locks = “cmd: perm(locks) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
assign a lock definition to an object

Usage:
  @lock <object>[ = <lockstring>]
  or
  @lock[/switch] object/<access_type>

Switch:
  del - delete given access type
  view - view lock associated with given access type (default)

If no lockstring is given, shows all locks on
object.

Lockstring is on the form
   access_type:[NOT] func1(args)[ AND|OR][ NOT] func2(args) ...]
Where func1, func2 ... valid lockfuncs with or without arguments.
Separator expressions need not be capitalized.

For example:
   'get: id(25) or perm(Wizards)'
The 'get' access_type is checked by the get command and will
an object locked with this string will only be possible to
pick up by Wizards or by object with id 25.

You can add several access_types after oneanother by separating
them by ';', i.e:
   'get:id(25);delete:perm(Builders)'
```

open source (building.py)

@mvattr (CmdMvAttr)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@mvattr“
- aliases = []
- locks = “cmd:perm(mvattr) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
move attributes between objects

Usage:
  @mvattr[/switch] <obj>/<attr> = <obj1>/<attr1> [,<obj2>/<attr2>,<obj3>/<attr3>,...]
  @mvattr[/switch] <obj>/<attr> = <obj1> [,<obj2>,<obj3>,...]
  @mvattr[/switch] <attr> = <obj1>/<attr1> [,<obj2>/<attr2>,<obj3>/<attr3>,...]
  @mvattr[/switch] <attr> = <obj1>[,<obj2>,<obj3>,...]

Switches:
  copy - Don't delete the original after moving.

Move an attribute from one object to one or more attributes on another
object. If you don't supply a source object, yourself is used.
```

open source (building.py)

@name (CmdName)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@name“
- aliases = [‘@rename‘]
- locks = “cmd:perm(rename) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
change the name and/or aliases of an object

Usage:
  @name obj = name;alias1;alias2

Rename an object to something new. Use *obj to
rename a player.
```

open source (building.py)

@open (CmdOpen)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@open“
- aliases = []
- locks = “cmd:perm(open) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
open a new exit from the current room

Usage:
  @open <new exit>[;alias;alias..][:typeclass] [,<return exit>[;alias;..
↪][:typeclass]]] = <destination>
```

```
Handles the creation of exits. If a destination is given, the exit
will point there. The <return exit> argument sets up an exit at the
destination leading back to the current room. Destination name
can be given both as a #dbref and a name, if that name is globally
unique.
```

open source (building.py)

@script (CmdScript)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@script“
- aliases = [‘@addscript‘]
- locks = “cmd:perm(script) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
attach a script to an object

Usage:
  @script[/switch] <obj> [= <script.path or scriptkey>]

Switches:
  start - start all non-running scripts on object, or a given script only
  stop - stop all scripts on objects, or a given script only

If no script path/key is given, lists all scripts active on the given
object.
Script path can be given from the base location for scripts as given in
settings. If adding a new script, it will be started automatically
(no /start switch is needed). Using the /start or /stop switches on an
object without specifying a script key/path will start/stop ALL scripts on
the object.
```

open source (building.py)

@set (CmdSetAttribute)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@set“
- aliases = []
- locks = “cmd:perm(set) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
set attribute on an object or player

Usage:
```

```
  @set <obj>/<attr> = <value>
  @set <obj>/<attr> =
  @set <obj>/<attr>
  @set *<player>/attr = <value>

Switch:
    edit: Open the line editor (string values only)

Sets attributes on objects. The second form clears
a previously set attribute while the last form
inspects the current value of the attribute
(if any).

The most common data to save with this command are strings and
numbers. You can however also set Python primities such as lists,
dictionaries and tuples on objects (this might be important for
the functionality of certain custom objects).  This is indicated
by you starting your value with one of |c'|n, |c"|n, |c(|n, |c[|n
or |c{ |n.
Note that you should leave a space after starting a dictionary ('{ ')
so as to not confuse the dictionary start with a colour code like \{g.
Remember that if you use Python primitives like this, you must
write proper Python syntax too - notably you must include quotes
around your strings or you will get an error.
```

open source (building.py)

@spawn (CmdSpawn)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@spawn“
- aliases = [‘spawn’]
- locks = “cmd:perm(spawn) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
spawn objects from prototype

Usage:
  @spawn
  @spawn[/switch] prototype_name
  @spawn[/switch] {prototype dictionary}

Switch:
  noloc - allow location to be None if not specified explicitly. Otherwise,
          location will default to caller's current location.

Example:
  @spawn GOBLIN
  @spawn {"key":"goblin", "typeclass":"monster.Monster", "location":"#2"}

Dictionary keys:
  |wprototype  |n - name of parent prototype to use. Can be a list for
```

```
                    multiple inheritance (inherits left to right)
  |wkey         |n - string, the main object identifier
  |wtypeclass   |n - string, if not set, will use settings.BASE_OBJECT_TYPECLASS
  |wlocation    |n - this should be a valid object or #dbref
  |whome        |n - valid object or #dbref
  |wdestination|n - only valid for exits (object or dbref)
  |wpermissions|n - string or list of permission strings
  |wlocks       |n - a lock-string
  |waliases     |n - string or list of strings
  |wndb_|n<name>  - value of a nattribute (ndb_ is stripped)
  any other keywords are interpreted as Attributes and their values.

The available prototypes are defined globally in modules set in
settings.PROTOTYPE_MODULES. If @spawn is used without arguments it
displays a list of available prototypes.
```

open source (building.py)

@tag (CmdTag)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@tag“
- aliases = [‘@tags‘]
- locks = “cmd:perm(tag) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
handles the tags of an object

Usage:
  @tag[/del] <obj> [= <tag>[:<category>]]
  @tag/search <tag>[:<category]

Switches:
  search - return all objects with a given Tag
  del - remove the given tag. If no tag is specified,
        clear all tags on object.

Manipulates and lists tags on objects. Tags allow for quick
grouping of and searching for objects.  If only <obj> is given,
list all tags on the object.  If /search is used, list objects
with the given tag.
The category can be used for grouping tags themselves, but it
should be used with restrain - tags on their own are usually
enough to for most grouping schemes.
```

open source (building.py)

@tel (CmdTeleport)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@tel“
- aliases = [‘@teleport‘]
- locks = “cmd:perm(teleport) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
teleport object to another location

Usage:
  @tel/switch [<object> =] <target location>

Examples:
  @tel Limbo
  @tel/quiet box Limbo
  @tel/tonone box

Switches:
  quiet  - don't echo leave/arrive messages to the source/target
           locations for the move.
  intoexit - if target is an exit, teleport INTO
             the exit object instead of to its destination
  tonone - if set, teleport the object to a None-location. If this
           switch is set, <target location> is ignored.
           Note that the only way to retrieve
           an object from a None location is by direct #dbref
           reference. A puppeted object cannot be moved to None.

Teleports an object somewhere. If no object is given, you yourself
is teleported to the target location.
```

open source (building.py)

@tunnel (CmdTunnel)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@tunnel“
- aliases = [‘@tun‘]
- locks = “cmd: perm(tunnel) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
create new rooms in cardinal directions only

Usage:
  @tunnel[/switch] <direction> [= roomname[;alias;alias;...][:typeclass]]

Switches:
  oneway - do not create an exit back to the current location
  tel - teleport to the newly created room

Example:
```

```
  @tunnel n
  @tunnel n = house;mike's place;green building

This is a simple way to build using pre-defined directions:
 |wn,ne,e,se,s,sw,w,nw|n (north, northeast etc)
 |wu,d|n (up and down)
 |wi,o|n (in and out)
The full names (north, in, southwest, etc) will always be put as
main name for the exit, using the abbreviation as an alias (so an
exit will always be able to be used with both "north" as well as
"n" for example). Opposite directions will automatically be
created back from the new room unless the /oneway switch is given.
For more flexibility and power in creating rooms, use @dig.
```

open source (building.py)

@typeclass (CmdTypeclass)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@typeclass“
- aliases = [‘@update‘, ‘@swap‘, ‘@type‘, ‘@parent‘]
- locks = “cmd:perm(typeclass) or perm(Builders)”
- help_category = “Building”
- __doc__ string (auto-help):

```
set or change an object's typeclass

Usage:
  @typclass[/switch] <object> [= <typeclass.path>]
  @type                      ''
  @parent                    ''
  @swap - this is a shorthand for using /force/reset flags.
  @update - this is a shorthand for using the /force/reload flag.

Switch:
  show - display the current typeclass of object (default)
  update - *only* re-run at_object_creation on this object
           meaning locks or other properties set later may remain.
  reset - clean out *all* the attributes and properties on the
           object - basically making this a new clean object.
  force - change to the typeclass also if the object
           already has a typeclass of the same name.
Example:
  @type button = examples.red_button.RedButton

If the typeclass.path is not given, the current object's
typeclass is assumed.

View or set an object's typeclass. If setting, the creation hooks
of the new typeclass will be run on the object. If you have
clashing properties on the old class, use /reset. By default you
are protected from changing to a typeclass of the same name as the
one you already have - use /force to override this protection.
```

```
The given typeclass must be identified by its location using
python dot-notation pointing to the correct module and class. If
no typeclass is given (or a wrong typeclass is given). Errors in
the path or new typeclass will lead to the old typeclass being
kept. The location of the typeclass module is searched from the
default typeclass directory, as defined in the server settings.
```

open source (building.py)

@unlink (CmdUnLink)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@unlink"
- aliases = []
- locks = "cmd:perm(unlink) or perm(Builders)"
- help_category = "Building"
- __doc__ string (auto-help):

```
remove exit-connections between rooms

Usage:
  @unlink <Object>

Unlinks an object, for example an exit, disconnecting
it from whatever it was connected to.
```

open source (building.py)

@wipe (CmdWipe)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@wipe"
- aliases = []
- locks = "cmd:perm(wipe) or perm(Builders)"
- help_category = "Building"
- __doc__ string (auto-help):

```
clear all attributes from an object

Usage:
  @wipe <object>[/attribute[/attribute...]]

Example:
  @wipe box
  @wipe box/colour

Wipes all of an object's attributes, or optionally only those
matching the given attribute-wildcard search string.
```

open source (building.py)

command (ObjManipCommand)

Belongs to the cmdset (“_).

- key = “command”
- aliases = []
- locks = “cmd:all()”
- help_category = “General”
- __doc__ string (auto-help):

```
This is a parent class for some of the defining objmanip commands
since they tend to have some more variables to define new objects.

Each object definition can have several components. First is
always a name, followed by an optional alias list and finally an
some optional data, such as a typeclass or a location. A comma ','
separates different objects. Like this:

    name1;alias;alias;alias:option, name2;alias;alias ...

Spaces between all components are stripped.

A second situation is attribute manipulation. Such commands
are simpler and offer combinations

    objname/attr/attr/attr, objname/attr, ...
```

open source (building.py)

comms.py

View Source

@cboot (CmdCBoot)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@cboot“
- aliases = []
- locks = “cmd: not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
kick a player from a channel you control

Usage:
   @cboot[/quiet] <channel> = <player> [:reason]
```

```
Switches:
   quiet - don't notify the channel

Kicks a player or object from a channel you control.
```

open source (comms.py)

@ccreate (CmdChannelCreate)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@ccreate“
- aliases = [‘channelcreate’]
- locks = “cmd:not pperm(channel_banned) and pperm(Players)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
create a new channel

Usage:
 @ccreate <new channel>[;alias;alias...] = description

Creates a new channel owned by you.
```

open source (comms.py)

@cdesc (CmdCdesc)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@cdesc“
- aliases = []
- locks = “cmd:not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
describe a channel you control

Usage:
  @cdesc <channel> = <description>

Changes the description of the channel as shown in
channel lists.
```

open source (comms.py)

@cdestroy (CmdCdestroy)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@cdestroy“
- aliases = []
- locks = “cmd: not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
destroy a channel you created

Usage:
  @cdestroy <channel>

Destroys a channel that you control.
```

open source (comms.py)

@cemit (CmdCemit)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@cemit“
- aliases = [‘@cmsg‘]
- locks = “cmd: not pperm(channel_banned) and pperm(Players)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
send an admin message to a channel you control

Usage:
  @cemit[/switches] <channel> = <message>

Switches:
  sendername - attach the sender's name before the message
  quiet - don't echo the message back to sender

Allows the user to broadcast a message over a channel as long as
they control it. It does not show the user's name unless they
provide the /sendername switch.
```

open source (comms.py)

@channels (CmdChannels)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@channels“
- aliases = [‘comlist’, ‘channellist’, ‘all channels’, ‘channels’, ‘@clist‘, ‘chanlist’]
- locks = “cmd: not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
list all channels available to you

Usage:
  @channels
  @clist
  comlist

Lists all channels available to you, whether you listen to them or not.
Use 'comlist' to only view your current channel subscriptions.
Use addcom/delcom to join and leave channels
```

open source (comms.py)

@clock (CmdClock)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@clock“
- aliases = []
- locks = “cmd:not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
change channel locks of a channel you control

Usage:
  @clock <channel> [= <lockstring>]

Changes the lock access restrictions of a channel. If no
lockstring was given, view the current lock definitions.
```

open source (comms.py)

@cwho (CmdCWho)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@cwho“
- aliases = []
- locks = “cmd: not pperm(channel_banned)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
show who is listening to a channel

Usage:
  @cwho <channel>

List who is connected to a given channel you have access to.
```

open source (comms.py)

@irc2chan (CmdIRC2Chan)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@irc2chan“
- aliases = []
- locks = “cmd:serversetting(IRC_ENABLED) and pperm(Immortals)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
link an evennia channel to an external IRC channel

Usage:
  @irc2chan[/switches] <evennia_channel> = <ircnetwork> <port> <#irchannel> <botname>
↪[:typeclass]
  @irc2chan/delete botname|#dbid

Switches:
  /delete     - this will delete the bot and remove the irc connection
                to the channel. Requires the botname or #dbid as input.
  /remove     - alias to /delete
  /disconnect - alias to /delete
  /list       - show all irc<->evennia mappings
  /ssl        - use an SSL-encrypted connection

Example:
  @irc2chan myircchan = irc.dalnet.net 6667 #mychannel evennia-bot
  @irc2chan public = irc.freenode.net 6667 #evgaming #evbot:players.mybot.MyBot

This creates an IRC bot that connects to a given IRC network and
channel. If a custom typeclass path is given, this will be used
instead of the default bot class.
The bot will relay everything said in the evennia channel to the
IRC channel and vice versa. The bot will automatically connect at
server start, so this command need only be given once. The
/disconnect switch will permanently delete the bot. To only
temporarily deactivate it, use the  |w@services|n command instead.
Provide an optional bot class path to use a custom bot.
```

open source (comms.py)

@rss2chan (CmdRSS2Chan)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@rss2chan“
- aliases = []
- locks = “cmd:serversetting(RSS_ENABLED) and pperm(Immortals)”
- help_category = “Comms”
- __doc__ string (auto-help):

```
link an evennia channel to an external RSS feed

Usage:
  @rss2chan[/switches] <evennia_channel> = <rss_url>

Switches:
  /disconnect - this will stop the feed and remove the connection to the
                channel.
  /remove     -                                    "
  /list       - show all rss->evennia mappings

Example:
  @rss2chan rsschan = http://code.google.com/feeds/p/evennia/updates/basic

This creates an RSS reader  that connects to a given RSS feed url. Updates
will be echoed as a title and news link to the given channel. The rate of
updating is set with the RSS_UPDATE_INTERVAL variable in settings (default
is every 10 minutes).

When disconnecting you need to supply both the channel and url again so as
to identify the connection uniquely.
```

open source (comms.py)

addcom (CmdAddCom)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "addcom"
- aliases = ['aliaschan', 'chanalias']
- locks = "cmd:not pperm(channel_banned)"
- help_category = "Comms"
- __doc__ string (auto-help):

```
add a channel alias and/or subscribe to a channel

Usage:
   addcom [alias=] <channel>

Joins a given channel. If alias is given, this will allow you to
refer to the channel by this alias rather than the full channel
name. Subsequent calls of this command can be used to add multiple
aliases to an already joined channel.
```

open source (comms.py)

allcom (CmdAllCom)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "allcom"
- aliases = []

- locks = "cmd: not pperm(channel_banned)"
- help_category = "Comms"
- __doc__ string (auto-help):

```
perform admin operations on all channels

Usage:
  allcom [on | off | who | destroy]

Allows the user to universally turn off or on all channels they are on,
as well as perform a 'who' for all channels they are on. Destroy deletes
all channels that you control.

Without argument, works like comlist.
```

open source (comms.py)

delcom (CmdDelCom)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "delcom"
- aliases = ['delaliaschan', 'delchanalias']
- locks = "cmd:not perm(channel_banned)"
- help_category = "Comms"
- __doc__ string (auto-help):

```
remove a channel alias and/or unsubscribe from channel

Usage:
   delcom <alias or channel>
   delcom/all <channel>

If the full channel name is given, unsubscribe from the
channel. If an alias is given, remove the alias but don't
unsubscribe. If the 'all' switch is used, remove all aliases
for that channel.
```

open source (comms.py)

page (CmdPage)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "page"
- aliases = ['tell']
- locks = "cmd:not pperm(page_banned)"
- help_category = "Comms"
- __doc__ string (auto-help):

```
send a private message to another player

Usage:
  page[/switches] [<player>,<player>,... = <message>]
  tell        ''
  page <number>

Switch:
  last - shows who you last messaged
  list - show your last <number> of tells/pages (default)

Send a message to target user (if online). If no
argument is given, you will get a list of your latest messages.
```

open source (comms.py)

general.py

View Source

access (CmdAccess)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "access"
- aliases = ['hierarchy', 'groups']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
show your current game access

Usage:
  access

This command shows you the permission hierarchy and
which permission groups you are a member of.
```

open source (general.py)

drop (CmdDrop)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "drop"
- aliases = []
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
drop something

Usage:
  drop <obj>

Lets you drop an object from your inventory into the
location you are currently in.
```

open source (general.py)

get (CmdGet)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "get"
- aliases = ['grab']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
pick up something

Usage:
  get <obj>

Picks up an object from your location and puts it in
your inventory.
```

open source (general.py)

give (CmdGive)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "give"
- aliases = []
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
give away something to someone

Usage:
  give <inventory obj> = <target>

Gives an items from your inventory to another character,
placing it in their inventory.
```

open source (general.py)

home (CmdHome)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "home"
- aliases = []
- locks = "cmd:perm(home) or perm(Builders)"
- help_category = "General"
- __doc__ string (auto-help):

```
move to your character's home location

Usage:
  home

Teleports you to your home location.
```

open source (general.py)

inventory (CmdInventory)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "inventory"
- aliases = ['i', 'inv']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
view inventory

Usage:
  inventory
  inv

Shows your inventory.
```

open source (general.py)

look (CmdLook)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "look"
- aliases = ['l', 'ls']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
look at location or object

Usage:
  look
  look <obj>
  look *<player>

Observes your location or objects in your vicinity.
```

open source (general.py)

nick (CmdNick)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "nick"
- aliases = ['@nick', 'nicks', '@nicks', 'nickname', 'alias']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
define a personal alias/nick

Usage:
  nick[/switches] <string> [= [replacement_string]]
  nick[/switches] <template> = <replacement_template>
  nick/delete <string> or number
  nick/test <test string>

Switches:
  inputline - replace on the inputline (default)
  object    - replace on object-lookup
  player    - replace on player-lookup
  delete    - remove nick by name or by index given by /list
  clearall  - clear all nicks
  list      - show all defined aliases (also "nicks" works)
  test      - test input to see what it matches with

Examples:
  nick hi = say Hello, I'm Sarah!
  nick/object tom = the tall man
  nick build $1 $2 = @create/drop $1;$2     - (template)
  nick tell $1 $2=@page $1=$2               - (template)

A 'nick' is a personal string replacement. Use $1, $2, ... to catch arguments.
Put the last $-marker without an ending space to catch all remaining text. You
can also use unix-glob matching:

    * - matches everything
    ? - matches a single character
    [seq] - matches all chars in sequence
    [!seq] - matches everything not in sequence

Note that no objects are actually renamed or changed by this command - your nicks
```

```
are only available to you. If you want to permanently add keywords to an object
for everyone to use, you need build privileges and the @alias command.
```

open source (general.py)

pose (CmdPose)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "pose"
- aliases = [':', 'emote']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
strike a pose

Usage:
  pose <pose text>
  pose's <pose text>

Example:
  pose is standing by the wall, smiling.
   -> others will see:
  Tom is standing by the wall, smiling.

Describe an action being taken. The pose text will
automatically begin with your name.
```

open source (general.py)

say (CmdSay)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "say"
- aliases = ["'",'"']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
speak as your character

Usage:
  say <message>

Talk to those in your current location.
```

open source (general.py)

whisper (CmdWhisper)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = "whisper"
- aliases = []
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
Speak privately as your character to another

Usage:
  whisper <player> = <message>

Talk privately to those in your current location, without
others being informed.
```

open source (general.py)

help.py

View Source

@help (CmdSetHelp)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@help"
- aliases = ['@sethelp']
- locks = "cmd:perm(PlayerHelpers)"
- help_category = "Building"
- __doc__ string (auto-help):

```
Edit the help database.

Usage:
  @help[/switches] <topic>[[;alias;alias][,category[,locks]] [= <text>]

Switches:
  edit - open a line editor to edit the topic's help text.
  replace - overwrite existing help topic.
  append - add text to the end of existing topic with a newline between.
  extend - as append, but don't add a newline.
  delete - remove help topic.

Examples:
  @sethelp throw = This throws something at ...
  @sethelp/append pickpocketing,Thievery = This steals ...
  @sethelp/replace pickpocketing, ,attr(is_thief) = This steals ...
  @sethelp/edit thievery
```

```
This command manipulates the help database. A help entry can be created,
appended/merged to and deleted. If you don't assign a category, the
"General" category will be used. If no lockstring is specified, default
is to let everyone read the help file.
```

open source (help.py)

help (CmdHelp)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "help"
- aliases = ['?']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
View help or a list of topics

Usage:
  help <topic or command>
  help list
  help all

This will search for help on commands and other
topics related to the game.
```

open source (help.py)

player.py

View Source

@charcreate (CmdCharCreate)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@charcreate"
- aliases = []
- locks = "cmd:pperm(Players)"
- help_category = "General"
- __doc__ string (auto-help):

```
create a new character

Usage:
  @charcreate <charname> [= desc]
```

```
Create a new character, optionally giving it a description. You
may use upper-case letters in the name - you will nevertheless
always be able to access your character using lower-case letters
if you want.
```

open source (player.py)

@color (CmdColorTest)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@color“
- aliases = [‘color’]
- locks = “cmd:all()”
- help_category = “General”
- __doc__ string (auto-help):

```
testing which colors your client support

Usage:
  @color ansi||xterm256

Prints a color map along with in-mud color codes to use to produce
them.  It also tests what is supported in your client. Choices are
16-color ansi (supported in most muds) or the 256-color xterm256
standard. No checking is done to determine your client supports
color - if not you will see rubbish appear.
```

open source (player.py)

@ic (CmdIC)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@ic“
- aliases = [‘@puppet‘]
- locks = “cmd:all()”
- help_category = “General”
- __doc__ string (auto-help):

```
control an object you have permission to puppet

Usage:
  @ic <character>

Go in-character (IC) as a given Character.

This will attempt to "become" a different object assuming you have
the right to do so. Note that it's the PLAYER character that puppets
characters/objects and which needs to have the correct permission!
```

```
You cannot become an object that is already controlled by another
player. In principle <character> can be any in-game object as long
as you the player have access right to puppet it.
```

open source (player.py)

@ooc (CmdOOC)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@ooc“
- aliases = [‘@unpuppet‘]
- locks = “cmd:pperm(Players)”
- help_category = “General”
- __doc__ string (auto-help):

```
stop puppeting and go ooc

Usage:
  @ooc

Go out-of-character (OOC).

This will leave your current character and put you in a incorporeal OOC state.
```

open source (player.py)

@option (CmdOption)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@option“
- aliases = [‘@options‘]
- locks = “cmd:all()”
- help_category = “General”
- __doc__ string (auto-help):

```
Set an account option

Usage:
  @option[/save] [name = value]

Switch:
  save - Save the current option settings for future logins.
  clear - Clear the saved options.

This command allows for viewing and setting client interface
settings. Note that saved options may not be able to be used if
later connecting with a client with different capabilities.
```

open source (player.py)

@password (CmdPassword)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@password“
- aliases = []
- locks = “cmd:pperm(Players)”
- help_category = “General”
- __doc__ string (auto-help):

```
change your password

Usage:
  @password <old password> = <new password>

Changes your password. Make sure to pick a safe one.
```

open source (player.py)

@quell (CmdQuell)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@quell“
- aliases = [‘@unquell‘]
- locks = “cmd:pperm(Players)”
- help_category = “General”
- __doc__ string (auto-help):

```
use character's permissions instead of player's

Usage:
  quell
  unquell

Normally the permission level of the Player is used when puppeting a
Character/Object to determine access. This command will switch the lock
system to make use of the puppeted Object's permissions instead. This is
useful mainly for testing.
Hierarchical permission quelling only work downwards, thus a Player cannot
use a higher-permission Character to escalate their permission level.
Use the unquell command to revert back to normal operation.
```

open source (player.py)

@quit (CmdQuit)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@quit“
- aliases = [‘quit’]

- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
quit the game

Usage:
  @quit

Switch:
  all - disconnect all connected sessions

Gracefully disconnect your current session from the
game. Use the /all switch to disconnect from all sessions.
```

open source (player.py)

@sessions (CmdSessions)

Belongs to the cmdset DefaultSession (SessionCmdSet).

- key = %22@sessions"
- aliases = []
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
check your connected session(s)

Usage:
  @sessions

Lists the sessions currently connected to your account.
```

open source (player.py)

look (CmdOOCLook)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "look"
- aliases = ['l', 'ls']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
look while out-of-character

Usage:
  look
```

```
Look in the ooc state.
```

open source (player.py)

who (CmdWho)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = "who"
- aliases = ['doing']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
list who is currently online

Usage:
  who
  doing

Shows who is currently online. Doing is an alias that limits info
also for those with all permissions.
```

open source (player.py)

system.py

View Source

@about (CmdAbout)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@about"
- aliases = ['@version']
- locks = "cmd:all()"
- help_category = "System"
- __doc__ string (auto-help):

```
show Evennia info

Usage:
  @about

Display info about the game engine.
```

open source (system.py)

@objects (CmdObjects)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@objects“
- aliases = [‘@listobjects‘, ‘@stats‘, ‘@db‘, ‘@listobjs‘]
- locks = “cmd:perm(listobjects) or perm(Builders)”
- help_category = “System”
- __doc__ string (auto-help):

```
statistics on objects in the database

Usage:
  @objects [<nr>]

Gives statictics on objects in database as well as
a list of <nr> latest objects in database. If not
given, <nr> defaults to 10.
```

open source (system.py)

@py (CmdPy)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@py“
- aliases = [‘!’]
- locks = “cmd:perm(py) or perm(Immortals)”
- help_category = “System”
- __doc__ string (auto-help):

```
execute a snippet of python code

Usage:
  @py <cmd>
  @py/edit

Switches:
  time - output an approximate execution time for <cmd>
  edit - open a code editor for multi-line code experimentation

Separate multiple commands by ';' or open the editor using the
/edit switch.  A few variables are made available for convenience
in order to offer access to the system (you can import more at
execution time).

Available variables in @py environment:
  self, me                  : caller
  here                      : caller.location
  ev                        : the evennia API
  inherits_from(obj, parent) : check object inheritance
```

```
You can explore The evennia API from inside the game by calling
evennia.help(), evennia.managers.help() etc.

|rNote: In the wrong hands this command is a severe security risk.
It should only be accessible by trusted server admins/superusers.|n
```

open source (system.py)

@reload (CmdReload)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@reload“
- aliases = []
- locks = “cmd:perm(reload) or perm(Immortals)”
- help_category = “System”
- __doc__ string (auto-help):

```
reload the server

Usage:
  @reload [reason]

This restarts the server. The Portal is not
affected. Non-persistent scripts will survive a @reload (use
@reset to purge) and at_reload() hooks will be called.
```

open source (system.py)

@reset (CmdReset)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@reset“
- aliases = [‘@reboot‘]
- locks = “cmd:perm(reload) or perm(Immortals)”
- help_category = “System”
- __doc__ string (auto-help):

```
reset and reboot the server

Usage:
  @reset

Notes:
  For normal updating you are recommended to use @reload rather
  than this command. Use @shutdown for a complete stop of
  everything.

This emulates a cold reboot of the Server component of Evennia.
The difference to @shutdown is that the Server will auto-reboot
```

```
and that it does not affect the Portal, so no users will be
disconnected. Contrary to @reload however, all shutdown hooks will
be called and any non-database saved scripts, ndb-attributes,
cmdsets etc will be wiped.
```

open source (system.py)

@scripts (CmdScripts)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@scripts"
- aliases = ['@listscripts', '@globalscript']
- locks = "cmd:perm(listscripts) or perm(Wizards)"
- help_category = "System"
- __doc__ string (auto-help):

```
list and manage all running scripts

Usage:
  @scripts[/switches] [#dbref, key, script.path or <obj>]

Switches:
  start - start a script (must supply a script path)
  stop - stops an existing script
  kill - kills a script - without running its cleanup hooks
  validate - run a validation on the script(s)

If no switches are given, this command just views all active
scripts. The argument can be either an object, at which point it
will be searched for all scripts defined on it, or a script name
or #dbref. For using the /stop switch, a unique script #dbref is
required since whole classes of scripts often have the same name.

Use @script for managing commands on objects.
```

open source (system.py)

@server (CmdServerLoad)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@server"
- aliases = ['@serverload', '@serverprocess']
- locks = "cmd:perm(list) or perm(Immortals)"
- help_category = "System"
- __doc__ string (auto-help):

```
show server load and memory statistics

Usage:
```

```
   @server[/mem]

Switch:
    mem - return only a string of the current memory usage
    flushmem - flush the idmapper cache

This command shows server load statistics and dynamic memory
usage. It also allows to flush the cache of accessed database
objects.

Some Important statistics in the table:

|wServer load|n is an average of processor usage. It's usually
between 0 (no usage) and 1 (100% usage), but may also be
temporarily higher if your computer has multiple CPU cores.

The |wResident/Virtual memory|n displays the total memory used by
the server process.

Evennia |wcaches|n all retrieved database entities when they are
loaded by use of the idmapper functionality. This allows Evennia
to maintain the same instances of an entity and allowing
non-persistent storage schemes. The total amount of cached objects
are displayed plus a breakdown of database object types.

The |wflushmem|n switch allows to flush the object cache. Please
note that due to how Python's memory management works, releasing
caches may not show you a lower Residual/Virtual memory footprint,
the released memory will instead be re-used by the program.
```

open source (system.py)

@service (CmdService)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@service“
- aliases = [‘@services‘]
- locks = “cmd:perm(service) or perm(Immortals)”
- help_category = “System”
- __doc__ string (auto-help):

```
manage system services

Usage:
  @service[/switch] <service>

Switches:
  list    - shows all available services (default)
  start   - activates or reactivate a service
  stop    - stops/inactivate a service (can often be restarted)
  delete - tries to permanently remove a service

Service management system. Allows for the listing,
```

```
starting, and stopping of services. If no switches
are given, services will be listed. Note that to operate on the
service you have to supply the full (green or red) name as given
in the list.
```

open source (system.py)

@shutdown (CmdShutdown)

Belongs to the cmdset DefaultPlayer (PlayerCmdSet).

- key = %22@shutdown“
- aliases = []
- locks = “cmd:perm(shutdown) or perm(Immortals)”
- help_category = “System”
- __doc__ string (auto-help):

```
stop the server completely

Usage:
  @shutdown [announcement]

Gracefully shut down both Server and Portal.
```

open source (system.py)

@time (CmdTime)

Belongs to the cmdset DefaultCharacter (CharacterCmdSet).

- key = %22@time“
- aliases = [‘@uptime‘]
- locks = “cmd:perm(time) or perm(Players)”
- help_category = “System”
- __doc__ string (auto-help):

```
show server time statistics

Usage:
  @time

List Server time statistics such as uptime
and the current time stamp.
```

open source (system.py)

unloggedin.py

View Source

__unloggedin_look_command (CmdUnconnectedLook)

Belongs to the cmdset DefaultUnloggedin (UnloggedinCmdSet).

OBS: This is a [[System Command|Commands]] name - it belongs to a number of command names the server calls in certain situations or as fallbacks.

- key = "__unloggedin_look_command"
- aliases = ['look', 'l']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
look when in unlogged-in state

Usage:
  look

This is an unconnected version of the look command for simplicity.

This is called by the server and kicks everything in gear.
All it does is display the connect screen.
```

open source (unloggedin.py)

connect (CmdUnconnectedConnect)

Belongs to the cmdset DefaultUnloggedin (UnloggedinCmdSet).

- key = "connect"
- aliases = ['co', 'conn', 'con']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
connect to the game

Usage (at login screen):
  connect playername password
  connect "player name" "pass word"

Use the create command to first create an account before logging in.

If you have spaces in your name, enclose it in double quotes.
```

open source (unloggedin.py)

create (CmdUnconnectedCreate)

Belongs to the cmdset DefaultUnloggedin (UnloggedinCmdSet).

- key = "create"

- aliases = ['cr', 'cre']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
create a new player account

Usage (at login screen):
  create <playername> <password>
  create "player name" "pass word"

This creates a new player account.

If you have spaces in your name, enclose it in double quotes.
```

open source (unloggedin.py)

help (CmdUnconnectedHelp)

Belongs to the cmdset DefaultUnloggedin (UnloggedinCmdSet).

- key = "help"
- aliases = ['h', '?']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
get help when in unconnected-in state

Usage:
  help

This is an unconnected version of the help command,
for simplicity. It shows a pane of info.
```

open source (unloggedin.py)

quit (CmdUnconnectedQuit)

Belongs to the cmdset DefaultUnloggedin (UnloggedinCmdSet).

- key = "quit"
- aliases = ['q', 'qu']
- locks = "cmd:all()"
- help_category = "General"
- __doc__ string (auto-help):

```
quit when in unlogged-in state

Usage:
  quit

We maintain a different version of the quit command
here for unconnected players for the sake of simplicity. The logged in
version is a bit more complicated.
```

open source (unloggedin.py)

Evennia Devel

This page serves as a changelog of the various bigger updates of Evennia over time.

Batchprocessor and misc updates Aug 2016

*Copied from the 'original mailing list post'_*

Changes to the batchcode processor usage

There are some changes to the Batchcode processor:

- The file path loading is a lot less finicky. A common error with using either of the batch processors was that people could not find their files by entering the python-style path to it. The reason this behaves differently is that Evennia converts this to an actual OS file path so it can be opened as a file rather than a Python module. In the past Evennia was trying to be a little too smart behind the scenes, by prepending the paths from settings.BATCHCODE_PATHS in front of whatever you used. This it will still do, but it will also be more accepting of people using full paths from the mygame or evennia folder's roots and some other tricks. In short, it just behaves more intuitively. This also modifies evennia.utils.utils.pypath_to_realpath, so if you are using this utility function you should check it out; it now also accepts a list of file prefix paths to try.
- The batchcode parser was completely overhauled. This closes several old bugs.
- The use of the batchcode processor to load a bog standard Python file is now considered standard, in the processor and in the documentation. The use of #HEADER, #CODE and #INCLUDE are considered optional extras for testing the code or stepping through it interactively.
- The #CODE block syntax has changed. The old #CODE name (varname1, varname2,,) syntax is gone and is just #CODE. If you want to run a script over and over and don't want it to create a pile of same-named objects when doing so, you should check for the global DEBUG variable made available to all batchcode scripts at execution (this made available regardless of #CODE or not):

```
# in batchcode script file

obj1 = create_object("key", ...)

if DEBUG: # only true if batchcode processor runs in /debug mode
    obj.delete()
```

Apart from the debug running, all old batchcode files will work the same (the #CODE string is just a comment to Python after all.

- The docs have been fully updated to reflect these changes.

New utils.justify

Added a new text utility, evennia.utils.utils.justify. This allows justifying and indenting full blocks of text. It also honors new paragraphs.

There was also a change to utils.wrap to make sure it indents on all lines, not skipping the very first line.

New @chardelete command

Players can now use the @chardelete command to themselves delete characters they have previously created with @charcreate. Deleting a character via some other means will also now no longer show a login traceback for users of MULTISESSION_MODEs > 1.

Evennia game index: List your Evennia game!

The Evennia game index (games.evennia.com) is considered stable rather than experimental now. It is however still pretty empty, mostly because people don't know about it yet. So don't be shy to register even your unfinished game with it - it's just a setting in your setting file. Your game does not need to be ready for prime time to register - you can choose to set it to pre-alpha status.

Spawner updates

- The Spawner was updated to now also accept the exec key - this allows devs to provide an executable snippet of code and is necessary for calling custom handlers on the spawned object etc. Use of exec is restricted to people able to write prototype files: Builders cannot supply exec arguments when creating protoype dicts from the command line with @spawn (this would be a security risk).
- All the prototype keys sent to the spawner can now also point to a callable taking no arguments. This allows for dynamic allocation of any property or attribute on the object, like random names, descriptions, stats etc etc.

Many bug fixes

A lot of bugs and issues have been closed lately. Including, but not exclusive to:

- Webclient handles ANSI backgrounds correctly now.
- RPsystem contrib has had a lot of fixing up as part of being used in Ainneve. It now adds a new say command and uses sdescs in more places.
- Scripts saw some fixes related to restarting
- Client-related compatibility fixes
- EvForm fixes (allows multi-character markup now)

Evennia 0.6 release May 2016 - OOB/webclient overhaul

Copied from 'original mailing list announcement'_.

- `msg()` now has a different callsign: `msg(text=None, **kwargs)`. All data being sent out is now considered an OOB (Out of band) instruction, including the default text (even though some protocols like telnet send it in a special way, it's still handled as an OOB instruction). These "sendcommands" are given as keyword arguments to the msg method and can be called either with a string or with a list of arguments or a tuple (args, kwargs) to be passed on to the protocol.

- `msg() accepts one special keyword and that is`options‘. This is a dictionary of options that will only affect the protocol sending. This is where you pass that you don’t want to parse ansi colors etc. So where you before would write msg(string, raw=True), you now need to do msg(string, options={“raw”:True}). The prompt is a separate OOB command, so you use msg(prompt=prompttext) as before.
- Inputfuncs - these are customizable functions you can now easily supply to manage any incoming command from the client. They are added in the same ways other plugins like lockfuncs or inlinefuncs are handled in Evennia - by simply defining them in a module Evennia then imports and uses. The default input is the text inputcommand, for which Evennia has a default handler (which launches the cmdhandler), but you could add support for any client-side instruction this way.
- OOB instructions for GMCP and MSDP have been included, as well as for Evennia’s JSON-based webclient instructions. These are just functions in a file that Evennia reads at runtime.
- Handles mudlet’s GMCP handshake (which is not standard except for IRE games)
- Added default commands for MSDP and bridges to GMCP.
- Protocol input/output signatures have changed a little. This is mainly of interest if you want to implement your own protocol though.
- The MonitorHandler in evennia/scripts/monitorhandler is a convenient mechanism for monitoring property- or Attribute-changes on any object. You assign a callback to the change which will trigger when anything changes. This is very useful to use with OOB instructions (like reacting and updating a health bar whenever the health attribute changes, for example).
- The TickerHandler has seen a lot of updates to make it more generally useful. You can now assign any function to it, to be ticked, not only methods on objects. This means that the callsign has changed quite a bit and users of the old TickerHandler will need to look into reworking their calls.
- The OOBHandler is no more since the OOB system is now integrated completely with the message sending system rather than being tacked on like before. All the functionality of the old OOBHandler is now handled between the MonitorHandler and the revamped TickerHandler.
- Added settings.IN_GAME_ERRORS. This is False by default. When active, it will echo Python errors not only to the log but also show the traceback in-game. This is useful for debugging but not good for a production game (where the user instead can report the log time stamp).
- The webclient has been completely reworked. It now consists of the evennia.js javascript library that handles all communication with the server. It also falls back to AJAX/comet and has a much improved handling of timeouts and various errors. This is meant to be used by whatever gui library is used.It handles the new OOB mechanism natively, allowing you to send and receive custom OOB instructions easily (e.g. from a custom GUI). The second component is evennia_gui.js which is the front-end of the client. This, together with the webclient.html and webclient.css files, implements the “telnet-like” default client. The idea is that his could be easily swapped out if someone wants to use another gui library for the front end. -The Webclient is a lot more stable and standards-compliant than before, and now also supports xterm256 colors like the other protocols.
- The evennia/web/ layout has been completely reworked so that there are now two easily understandable website/ and webclient/ folders rather than having all files spread around in the root of that folder. The “prosimii” template is still used but is now renamed to “website” (credit remains) to make it more obvious to remember and overload.
- Added the ability to overload the default Command parent from your game, for making sweeping changes to commands
- Updated the ChannelHandler:
- Added the ability to customize the Channel command.
- Also added an automatic log (to a log file) of all channels, along with a /history switch to the channel that allows for viewing past channel posts.

- Only the nested inlinefuncs on the form `$func()` are now available. The already deprecated {func ... {/func style inline funcs have been removed.
- The @options command now allows a player to manually set a wide range of options related to their connection. This helps for cases when Evennia cannot accurately identify your client, or you want to use some other setting than is default for that client.
- The EvMenu can now also be set to be persistent across server reboots.
- The menu_login contrib now uses EvMenu.
- Added evennia –initsettings for adding a new, empty settings file to a game directory. This is useful when sharing game dirs between multiple developers.
- A lot of other bug fixes and small improvements all over.

New Library layout and typelcasses March 2015

*Copied from the original announcement here

Devel branch merged March 1, 2015

Changing typeclasses to use proxy models

To understand this change you need to be a little familiar with how typeclasses work in master. A typeclass is a Python class whose **setattr** and **getattributes** methods are overridden in such a way that whenever you store something on it, it actually stores that data on its connected database model instance. That database model instance is in turn cached and held in memory by Evennia. It stores a text string holding the python path to the typeclass associated with it. For example, when you @create a "Rock" object of the typeclass "src.objects.objects.Object", what happens is that a new database object (model ObjectDB) is created to represent this new line in the database table. This model then initialize an instance of its typeclass (src.objects.objects.Object in this example) and ties to it using its .typeclass property. Vice-versa, this new typeclass ties back to the database model instance via its .dbobj property. By simply storing different typeclass-paths in the database one can this way represent any number of different game entities using very few database models.

A drawback with the typeclass system in the master branch is that it introduces some custom limitations on its classes. Notably you need to use the create_object, create_script etc functions to create typeclassed objects (you cannot overload **init**) since the database model and typeclass must be tied together and initialized correctly at creation. Furthermore there is a constant exchange between the two objects with lookups on the typeclass often leads to the lookup on its dbobj and also the other way around - this introduces a small but noticeable performance hit over time. Furthermore, since django knows nothing about this typeclass business querying objects in the database deals with database models and not typeclasses (this is a small limitation since we implement our own managers and there are also suggested other means of modifying queries to address this particular issue). When it comes to understanding the inheritance of typeclasses this is also a bit cumbersome since the typeclass inheritance tree is not actually stored in the database and can thus not be searched easily through it (such as when wanting to find all objects of a given typeclass and its children).

Enter proxy models. Proxy models has been around in django for a while but they were not there when I first started with Evennia and they had slipped under my radar until user Volund made me aware of them in chat. At the time I was working on another typeclass revamp using the existing system - I threw that aaway after looking into proxy models. A proxy model is basically described in the django docs as a way to expand a django model with alternative Python code without needing to change the database schema - in short a proxy class using the database model for storage. Sounds familiar? As it turns out django's proxy models suit our typeclass needs very well, without the need of our custom overlying implementation. So during Christmas I have converted Evennia's typeclass system to use proxy models. Below are the advantages and features of this new typeclass implementation:

- typeclasses now inherit directly from their database models. All such models are automatically treated as a proxy (this is implemented using some (if I may say so) pretty sophisticated metaclass magic, so creating a new typeclass requires no extra boiler plate except inheriting from the right parent.
- There is no longer any difference between the database model and the typeclass (we still call the proxy children of the model "typeclasses" though). This means that the .dbobj and .typeclass properties does not make sense any more and were removed. All methods were moved from the database model to the typeclass and are now available directly through inheritance, so you should have much less need to go to the database model than you had. You can however still reach a typeclass' parent model at any time using typeclass.**dbclass**, which is the new standard along with the built-in **class** property. - - You can now query typeclasses directly, instead of only the main model. For example, if you have a typeclass "Rock" you can do Rock.objects.all() to get all the rocks. Conversely, if you do ObjectDB.objects.all() (or Rock.**dbclass**.objects.all()) you will (same as before) get all ObjectDB-derived instances, independent of typeclass. There are now also the new all_family, filter_family and get_family manager methods that allow you to query the database directly for an typeclass and all its subclasses, such as Rock.objects.all_family(). -
- You can now create new instances of typeclasses using normal initialization. So rock = Rock() will now get you a correct typeclass (you need to do rock.save() to actually store it, same as any django model). Through internal signalling saving this will still trigger the correct startup hooks. It should be noted that the create_* functions still offers more functionality, since they can accept more arguments and add things like permissions at creation time (using the plain construction you'd need to add such things manually). - - Interestingly, the new system requires no changes to the database schema, so the actual change needed in your code is not so big as one might think. -
- The new typeclass system is a lot easier to explain and should also be more efficient. Furthermore, it should be easier to cache using one of the many cache solutions available to Django (such as memcached) or using threading for supported database (more testing is needed of this though).
- There are two caveats of the new typeclass system: - -
- Django's proxies does not *quite* fit our needs. I have modified the query system to return typeclasses rather than database models. I have also introduced a django patch to allow proxies multiple inheritance as long as they all stem from the same model. This patch is included in Evennia and introduced transparently, but it should hopefully soon be a part of future django versions so we don't need this hack. -
- There can only exist one proxy model of a given name for a given base model. This means that we can no longer do stuff like "from src.objects.objects import Object as BaseObject" and then create a class Object(BaseObject) (as was done in the gamesrc/ example files). Django interprets these as two proxy models based off ObjectDB, both named "Object", something which is not allowed. For this reason the default typeclasses are now called DefaultObject, DefaultRoom, DefaultExit etc, to allow end users the possibility to use the shorter Object, Exit, Room etc. This is one reason (apart from legacyy) that the classes are still called "typeclasses".
- So what changes does the new typeclass system require from you, the end user. Surprisingly little. The main thing is expected to be to remove your use of .dbobj and .typeclass and to change eventual imports of the default Object, Exit etc to instead be named DefaultObject, DefaultExit e

. . . But I'm not done yet. what WILL require some more changes is the next new change. Read on . . . - - -

Evennia becomes a library - -

Once I did the typeclass revamp I though I could just as well continue and add the other big change that has been discussed for a long time - converting Evennia to a proper package/library structure. The library change is also operational in devel now. - - The package change means that Evennia itself takes on the role of a library with an "evennia" executable to do operations. The "game" directory is no longer shipped with the system but is created on-the-fly for each new game using the evennia launcher. This allows the evennia library to exist completely separately from the game implementation in the same way as django does. The freshly created game folder has empty starting modules for the common game entities, log files and configs and a dynamically created settings.py file that links to

those files. So no more "copy example/cmdset.py up one level, then change your setting file to point to it . . . " as we use in master. Since the game directory is created on the fly it is not a part of Evennia's version control which means that you can change it and restructure it as you please without being afraid of running into merge conflicts down the line. You can also easily create multiple games in different folders (as long as you change their ports to avoid collisions).

Here is an example of creating a new game with evennia once the library has been installed:

```
evennia init mygame
cd mygame -
evennia migrate         # Creates the database based on your (possibly tweaked)␣
↪settings.  -
evennia -i start        # start the server - - -
```

Main differences when coding using the new Evennia library: -

- src/ is no more. The src/ folder has been renamed evennia/ and the library is expected to be imported simply as "evennia" in your code. -
- ev.py is no more. All of the flat API has been included in evennia.**init**, which means that you can get to most common things directly via just the evennia import (such as evennia.DefaultObject). -
- game/ is no more, obviously. This is now the dynamically-created folder. The old examples, such as the red button has been moved to a new contrib/tutorial_examples/ folder. -
- game/manage.py was merged with game/evennia.py into the new bin/evennia executable. For now, call it explicitly with python path-to-evennialib/evennia. This will need to be made automatically available on $PATH down the line and linux users can do so manually if they want.
- the default typeclass paths have changed to be located in the new game dir rather than in the evennia source tree. Migrations for this are not yet finished so use a fresh database to test. - - - So why this change? The main advantage (and goal) of the package restructure is that this makes it easier to distribute Evennia in a more accessible form. Once we have worked out the kinks, it means that we can distribute Evennia in pypi and that those of you who are not interested in git will be able to do something like "pip install evennia" without much fuzz. Also getting evennia into other package systems (like debian) should be easier. It will also lead to Evennia adopting a more formal release schedule with version numbers (more on this in the future). Cool cats will of course still be able to follow and help using the bleeding edge git version as before. -

Memory optimizations of June 2014

Text from original announcement. See also the Devblog post.

To understand what was done, here is a little background. Python keeps tracks of all objects (from variables to classes and everything in between) via a memory reference. When other objects reference that object it tracks that too. Some objects don't need to be in memory (because noone can access them any more), so Python's garbage collector goes through them and cleans such objects up so memory can be used for other things. The garbage collector will not do so however if some other object (which will not be garbage-collected) still has a reference to the object. This is what you want - you don't want existing objects to stop working because an object they rely on is suddenly not there.

Evennia uses something called the idmapper. This is a cache mechanism that allows objects to only be loaded from database once and be reused when later accessed. The speedup achieved from this is significant, but it is also a critical part of the typeclass system - if the memory representation changed all the time we could not store things like non-persistent attributes and would have to re-initialize all cmdhandlers, attributehandlers, lockhandlers and what have you every time you accessed an object.

The tradeoff of speed and utility is memory usage. Since the idmapper keeps those references, memory usage of Evennia could rise rapidly with an increasing number of objects.

Whereas some objects (with temporary attributes) should indeed not be garbage collected, in a working game there is likely to be objects without such volatile data that are not used some of the time - simply because players or the game don't go there for the moment. For such objects it may be okay to re-load them on demand rather than keep them in memory when not needed.

When looking into this I found that simply flushing the idmapper did not clean up all objects from memory. The reason for this has to do with Evennia holding other references.

So I went through a rather prolonged spree of cleanups where I gradually (and carefully) cleaned up Evennia's object referencing to a point where the only external reference to most objects were the idmapper cache reference. Removing that will now make the object possible to garbage-collect.

This is how the reference map used to look for an ObjectDB object before. Note the several references into the ObjectDB and the cyclic references for all handlers.

This is how the reference map looks now. The **instance** cache is the idmapper reference. There are also no more cyclic references for handlers (the diplay don't even pick up on them for this depth of display). Just removing that single link will now garbage-collect ObjectDB and its typeclass (ignore the g reference, that is just the variable holding the object ipython). We also see that the dbobj.typeclass <-> typeclass.dbobj references keep each other alive and when one goes the other one does too.

What will generally not be cleaned currently are objects with cmdsets on them. This is a forest of references that I might look into straightening at some point, but many such objects, like Characters and Players, should never be garbage collected anyway. Assigning a non-persistent Attribute via the ndb handler will make sure this object will not be cleaned out.

There are two ways to flush the idmapper cache in the latest push: manually or automatically. Manually it can be done via @server/flushmem. There is now a new global Script that will check the memory usage every 5 minutes and flush the cache if it exceeds a given value. The flush limit for this is defined by settings.IDMAPPER_CACHE_MAXSIZE. The value you need to set for this depends very much on the size of your game and the kind of usage you expect (notably how many players and how often objects need to be in memory). src/settings_defaul.py has a table listing the suggested size for holding various numbers of objects in cache. You probably won't need to mess with this until you run a production server.

Status as of May 2014

We are in a stretch of fixing bugs and optimizing. There is work ongoing of limiting the memory footprint of Evennia, this has not yet merged with master.

ANSIString/Evtable/Evform push of Feb-April 2014

This saw a series of work by contributor Kelketek on implementing a subclass of strings that can handle ANSI markers in a transparent way. Once this merged, the EvTable and EvForm modules became more generally useful and is now slowly replacing the old third-party PrettyTable (although prettytable will likely remain as a backup).

Github move in January 2014

*Text copied from original mailing list announcement from Jan 26. See also The Devblog post for a detailed account of the move.

As of today, Evennia's code, documentation and issue handling has officially moved over to GitHub, to http://github/evennia/evennia.

All links and feeds on the main evennia.com page has been changed to point to the new location. Mailing list and blog are not affected but the Commit mailing list is currently not working, if you are finding the Commit mailing list indispensable, reply here if I should put work into coercing github to send to it.

Practically, this means that Google Code's mercurial repository and wiki will no longer be updated. So to get updates you need to use the new github host. See our new GettingStarted page for updated info on how to get Evennia.

There were some cleanup of the Mercurial repository to make it convert cleanly over to GIT. So using conversion tools on your own repos may be a painful experience. If you followed guidelines and only made your local changes in game/gamesrc, the fastest and cleanest way for you to get going is to do make a new fresh clone of Evennia from github (or even better, fork it on Github), then just manually copy & paste your gamesrc changes (as well as game/settings.py and the database file game/evennia.db3 if you use SQLite3). Things should work normally from there. See our new Version Control wiki page for more info on using GIT and contributing to Evennia.

Devel-clone as of October 2013

This update focused on moving the webserver into Server as well as functioning OOB and reworked Attributes and Tags. Channels became Typeclassed.

*This clone has **not* yet merged with main. This text is copied from the mailing list post.*

New features

These are features that either don't affect existing APIs or introduce new, non-colliding ones.

- The webserver was moved from Portal into Server, for reasons outlined in earlier posts.
- Out-Of-Band (OOB) functionality. This uses the MSDP protocol to communicate with supported third-party clients (the webclient does not currently support OOB). The new OOBhandler supports tracking of variables and most of the default commands recommended by the MSDP protocol. GMCP support is not part of this update. From the API side, it means the msg() method have a new keyword 'oob', such as msg(oob=("send",{"key":"val"})
- Comm Channels are now Typeclassed entities. This means they can be customized much more than before using hooks and inheritance. src.comms.comms.py contains the new default channel typeclass and hooks. Settings. DEFAULT_COMM_TYPECLASS define the default typeclass.
- Most database field wrappers have been moved into the SharedMemoryObject metaclass. This makes the handling of database fields consistent and also makes the source code of models considerably shorter with less boiler plate. All database fields are updated individually now instead of having to save the entire database object every time a field changes. The API is otherwise unchanged - you still use obj.key="name" to save to the obj.db_key database field, for example. A new feature is that you can now give dbrefs to fields holding objects in order to store that object in the field. So self.location = "#44" should work.
- Attributes have three new fields: data, strvalue and category. All are optional. The first can be used for arbitrary string data (it is used by nick for the nick replacement). The second field, strvalue, is used for storing a value known to always be a string (as opposed to the normal value field which is pickled). This offers easier optimization and makes Attributes useful for more things. Category can be used to group Attributes (for example when they are used as Nicks by the nickhandler). Normal operations are not affected. Attributes are also now stored as a m2m fields on objects rather than via a reverse lookup.
- obj.tags is a new handler on all typeclassed objects. A Tag is unique and indexed and can be attached to any number of objects. It allows to tag and group any entity/entities for quick lookup later. Like all handlers you use get/add/remove/clear/all to manipulate tags.
- obj.nicks works similarly to before but it uses Attributes under the hood (using strvalue and data fields for nick replacement and category to determine which type of replacement to do).

- Sessions can also have their own cmdsets when the player has logged in. There are a few other new settings in settings_default, notably related to OOB and caching.
- New, reworked cache system.

Deprecations

These are features that have changed but where the old way still works - for now.

- Attributes are handled by the attributehandler (obj.attributes or obj.db), which means that the old on-object methods are all deprecated. Use of an deprecated method will result in a DeprecationWarning in your log. Note that obj.db works the same as before, it can (and should) replace all of these unless you are looking to operate on an Attribute you don't know the name of before execution.
- obj.has_attribute(attrname) -> obj.attributes.has(attrname)
- obj.get_attribute(attrname) -> obj.attributes.get(attrname)
- obj.set_attribute(attrname, value) -> obj.attributes.add(attrname, value)
- obj.del_attribute(attrname) -> obj.attributes.remove(attrname). There is also obj.attributes.clear() to remove all Attributes from obj.
- obj.get_all_attributes() -> obj.attributes.all()
- obj.secure_attr(attrname) -> obj.attributes.get(attrname, accessing_obj=aobj, default_access=True). The new get/set/remove/clear/all methods have these optional keywords to turn it into an access check. Setting default_access=False will fail the check if no accessing_obj is given.
- obj.attr() - this was just a wrapper for the above commands, use the new ones instead.
- obj.nattr() is replaced by the obj.nattributes handler instead. obj.ndb works the same as before. The usage of Aliases as 'tags' alluded to in the tutorials (e.g. for zones) should now be handled by Tags instead, they are intended for this purpose.

Incompatibilities

These are features/APIs that have changed to behave differently from before. Using the old way will lead to errors.

- Minimum Django version was upped from 1.4 to 1.5.
- User+PlayerDB -> PlayerDB. This means that django.contrib.auth.models.User is no longer used and all references to it should be changed to src.players.models.PlayerDB, which now holds all authorization information for a player account. Note that not all 3rd party Django apps have yet updated to allow a custom User-model. So there may be issues there (one such app known to have issues is DjangoBB).
- msg(text, data=None) has changed its API to `msg(text=None,` args, **kwargs)'. This makes no difference for most calls (basically anything just sending text). But if you used protocol options, such as msg(text,data={"raw":True}) you should now instead use msg(text, raw=True).
- obj.permissions="perm" used to add "perm" to a hidden list of permissions behind the scenes. This no longer works since permissions is now a full handler and should be called like this: obj.permissions.set("perm"). The handler support the normal get/add/remove/all as other handlers. Permissions now use Tags under the hood.
- obj.aliases="alias" used to add 'alias' to a hidden handler. This no longer works as obj.aliases is now a full handler: obj.aliases.set("alias"). This works like other handlers. Aliases now use Tags under the hood.
- All portal-level modules have moved from being spread out all over src.server into a new sub-folder src.server.portal. Change your imports as required.

- The default search/priority order for cmdsets have changed now that Sessions may also have cmdsets. Cmdsets are merged in the order session-player-puppet, which means that the puppet-level cmdset will default to overiding player-level cmdsets which in turn overrides session-level ones.
- Messages (using the msg() method) used to relay data puppet->player->session. Now, puppet-level relays data directly to the session level, without passing the player-level. This makes it easier to customize msg at each respective level separately, but if you overloaded player.msg() with the intent to affect all puppeted objects, you need to change this.
- If you used src.server.caches for anything (unlikely if you are not a core dev), the APIs of that has changed a lot. See that module.

Known Issues

- Whereas this merge will resolve a number of Issues from the list, most fixed ones will be feature requests up to this point. There are many known Issues which have not been touched. Some may be resolved as a side effect of other changes but many probably won't. This will come gradually. The wiki is of course also not updated yet, this will likely not happen until after this clone has been merged into main branch. For now, if you have usage questions, ask them here or on IRC.

Devel clone as of May 2013

_This update centered around making a player able to control multiple characters at the same time (the multplayer_mode=2 feature).*

- This clone was merged with main branch. This text is copied from the mailing list post._

Things you have to update manually:

If you have partially overloaded and import the default cmdsets into game/gamesrc, you have to update to their new names and locations:

- src.commands.default.cmdset_default.DefaultCmdSet changed name to src.commands.default.cmdset_character.CharacterCmdSe
- src.commands.default.cmdset_ooc.OOCCmdSet changed name to src.commands.default.cmdset_player.PlayerCmdSet (in the same way ev.default_cmds now holds CharacterCmdSet and PlayerCmdSet instead of the old names)

Note that if you already named your own cmdset class differently and have objects using those cmdsets in the database already, you should keep the old name for your derived class so as to not confuse existing objects. Just change the imports. The migrations will detect if any objects are using the old defaults and convert them to the new paths automatically.

Also the settings file variable names have changed:

- settings.CMDSET_DEFAULT has changed to settings.CMDSET_CHARACTER
- settings.CMDSET_OOC has changed to settings.CMDSET_PLAYER The system will warn you at startup if your settings file contains the old names.

If you have extensively modified Object Typeclasses, you need to update your hooks:

- obj.at_first_login(), at_pre_login(), at_post_login() and at_disconnect() are removed. They no longer make sense since the Player is no longer auto-tied to a Character (except in MULTISESSION_MODE=0 and 1 where this is retained as a special case). All "first time" effects and "at login" effects should now only be done on the same-named hooks on the Player, not on the Character/Object.

- New hooks on the Object are obj.at_pre_puppet(player), at_post_puppet(), at_pre_unpuppet() and at_post_unpuppet(player). These are now used for effects involving the Character going "into" the game world. So the default move from a None-location (previously in at_pre_login()) is now located in at_pre_puppet() instead and will trigger when the Player connects/disconnects to/from the Object/Character only.

  The Permission Hierarchy lock function (perm) has changed in an important way:

- Previously, the perm() lock function checked permission only on the Character, even if a Player was connected. This potentially opens up for escalation exploits and is also rather confusing now that the Player and Character is more decoupled (which permission is currently used?)
- perm() now checks primarily the Player for a hierarchy permission (Players, Builders, Admins etc, the stuff in settings.PERMISSION_HIERARCHY). Other types of permissions (non-hierarchical) are checked first against Player and then, if the Player does not have it, on the Character.
- The @quell command was moved from a contrib into the main distribution. It allows Players to force hierarchical permission checks to only take the currently puppeted Character into account and not the Player. This is useful for staff testing features with lower permissions than normal. Note that one can only downgrade one's Player permission this way - this avoids Player's escalating their permissions through controlling a high-perm Character. Superusers can never be quelled, same as before.

  This is not a show-stopper, but nevertheless an important change:

- settings.ALLOW_MULTISESSION was removed and is now replaced with MULTISESSION_MODE which can have a value of 0, 1 or 2.

Other Changes to be aware of

- Many-Characters-per-Player multisession mode. See the previous post here.
- Player.character does still exist for backwards compatability but it is now only valid in MULTISESSION_MODE 0 or 1. Also this link will be meaninless when the Player goes OOC - the Player-Object link is now completely severed (before it remained). For MULTISESSION_MODE=2, you must use Player.get_character(sessid). See src.commands.default.player.py for details on how to get the Character now.
- The @ic and @ooc and @ooclook commands use an Attribute `_playable_characters` to store a list of "your" characters. This is not hard-coded but only used by those commands. This is by default only used for listing convenience - locks are now the only thing blocking other users from puppeting your characters when you are not around. Keeping a list like this is now the only safe way to relate Characters with a given Player when that Player is offline.
- Character typeclass has new hooks at_pre_puppet
- ObjectDB.search() has a changed api: search(ostring, global_search=False, use_nicks=False, typeclass=None, location=None, attribute_name=None, quiet=False, exact=False. The changes here are the removal of the global_dbref keyword and that ignore_errors keyword was changed to quiet. More importantly the search function now always only return Objects (it could optionally return Players before). This means it no longer accepts the `*playername` syntax out of the box. To search for Players, use src.utils.search.player_search (you can always look for the asterisk manually in the commands where you want it). This makes the search method a lot more streamlined and hopefully consistent with expectations.
- object.player is now only defined when the Player is actually online (before the connection would remain also when offline). Contrary to before it now always returns a Player typeclass whenever it's defined (Issue 325)
- object.sessid is a new field that is always set together with character.player.
- object.msg() has a new api: msg(self, message, from_obj=None, data=None, sessid=0). In reality this is used mostly the same as before unless wanting to send to an unexpected session id. Since the object stores the sessid of the connected Player's session, leaving the keywords empty will populate them with sensible defaults.

- player.msg() also has changed: msg(self, outgoing_string, from_obj=None, data=None, sessid=None). The Player cannot easily determine the valid sessid on its own, so for Player commands, the sessid needs to be supplied or the msg will go to all sessions connected to the Player. In practice however, one uses the new Command.msg wrapper below:
- command.msg is a new wrapper. It's call api looks like this: msg(self, msg="", to_obj=None, from_obj=None, data=None, sessid=Noneall_sessions=False). This will solve the problem of having to remember any sessids for Player commands, since the command object itself remembers the sessid of its caller now. In a Player command, just use self.msg(string). To clarify, this is just a convenience wrapper instead of calling self.caller.msg(string, sessid=self.sessid) - that works identically but is a little more to write.
- The prettytable module is now included with Evennia. It was modified to handle Evennia's special ANSI color markers and is now the recommended way to output good-looking ASCII tables over using the old src.utils.format_table (which is still around)

Other changes

- New internal Attribute storage, using PickledFields rather than a custom solution; this now also allows transparent lookups of Attribute data directly on the database level (you could not do this (easily) before since the data is internally pickled).
- Updated all unittests to cover the default commands again, also with a considerably speedup.
- Plenty of cleanups and bug fixes all over
- Removed several deprecation warnings from moving to Django 1.4+ and a few others.
- Updated all examples in game/gamesrc and the various APIs

Status update as of December 2012

Mostly bug fixes and various cleanup this update. This is copied from the mailing list post.

Latest pushes to the repository fixes a few things in the Tutorial world. Notably the torch/splinter will light properly again now - which means you will be not be forever entombed under ground. Also I sometimes found that I couldn't solve the final puzzle. This is now fixed and you will now again be able to finish your quest by wreaking some well-deserved vengeance on that pesky Ghostly Apparition. I hadn't looked at the tutorial in a while which revealed a bunch of other small inconsistencies in how the Character was cleaned up afterwards, as well as some other small things, all now fixed. The tutorial world is meant to be a nice first look into what Evennia can do, so if you do come across further strangeness in it, don't be shy to report it. Also, it may be worth lingering on the west half of the swaying bridge longer than you should, just to see what happens.

In other news, there is now a "give" command in the default cmdset; it's very simple (for example the receiver have no choice but to accept what is given to them) but it helped debug the Tutorial world and is a neat command to build from anyway.

If you didn't notice, the latest changes places more strict regulation on how to reference database references from the default cmdset. Before you could do things like "ex 2" and expect to get Limbo. You will now have to do "ex #2", allowing objects to have numbered names as well (this was a feature request). The upshot is that the explicit dbref-search can be made global whereas key-searches can remain local. This is handled by a new keyword to object.search called "global_dbref". This means you can do things like "ex #23" and examine the object with dbref=23 wherever it is in the game. But you can also do "ex north" and not get a multi-match for every north exit in the game, but only the north in your current location. Thanks to Daniel Benoy for the feature request suggesting this. There might be more build commands were this is useful, they will be updated as I come across them or people report it.

Status update as of October 2011

This was an update related to the changes to persistence and other things on the docket. This text is copied from the mailing list post.

Here are some summaries of what's going on in the Evennia source at the moment:

Admin interface

The admin interface backend is being revamped as per issue 174. Interface is slowly getting better with more default settings and some pointless things being hidden away or given more sensible labels. It's still rough and some things, like creating a new Player is hardly intuitive yet (although it does work, it requires you to create three separate models (User-Player-Character) explicitly at this point). I'm also seeing a bunch of formatting errors under django1.3, not sure if this is media-related or something fishy with my setups, not everyone seems to see this (see issue 197 if you want to help test).

FULL_PERSISTENCE setting

... is no more. FULL_PERSISTENCE=True is now always in effect. The feature to activate this setting was added at a time when the typeclass system's caching mechanism was, to say the least, wasteful. This meant that many problems with FULL_PERSISTENCE=False were hidden (it "just worked" and so was an easy feature to add). This is no longer the case. It's not worth the effort to support the False setting in parallel. Like before you can still assign non-persistent data by use of the ndb operator.

Typeclass handling

Typeclasses are handled and managed and cached in a better way. Object.typeclass now actually returns the full instantiated typeclass object, not its class like before (you had to manually initiate it like dbobj.typeclass(dbobj)). The main reason for this change is that the system now allows very efficient calls to hook methods. The at_init() hook will now be called whenever any object is inititated - and it's very efficient; initiation will only happen whenever an entity is actually used in some ways and thus being cached (so an object in a seldomly-visited room might never be initiated, just as it should be).

Support for out-of-band communication

Nothing is done in the server with this yet, but I plan to have a generalized way to implementing out-of-band protocols to communicate with custom clients, via e.g. GMCP or MCP or similar. There are some efforts towards defining at least one of those protocols behind the scenes, but time will tell what comes of it.

Devel branch as of September 2011

This update concerned the creation of the Server/Portal structure.

This update has been merged into main. The text is copied from the mailing list post.

- Evennia was split into two processes: Server and Portal. The Server is the core game driver, as before. The Portal is a stand-alone program that handles incoming connections to the MUD. The two communicate through an AMP connection.

- Due to the new Portal/Server split, the old reload mechanism is no more. Reloading is now done much more efficiently - by rebooting the Server part. Since Players are connected to the Portal side, they will not be disconnected. When Server comes back up, the two will sync their sessions automatically. @reload has been fixed to handle the new system.
- The controller script evennia.py has been considerably revamped to control the Portal and Server processes. Tested also on WinXP. Windows process control works, but stopping from command line requires python2.7. Restarting from command line is not supported on Windows (use @restart from in-game).
- Courtesy of user raydeejay, the server now supports internationalization (i18n) so messages can be translated to any language. So far we don't have any languages translated, but the possibility is there.
- @reload will not kill "persistent" scripts and will call *at_server_reload()* hooks. New @reset command will work like an old server shutdown except it automatically restarts. @shutdown will kill both Server and Portal (no auto-restart)
- Lots of fixes and cleanup related to fixing these systems. Also the tutorial_world has seen some bugs fixed that became more obvious with the new reload system.
- Wiki was updated to further explain the new features.

Update as of May 2011

This update marks the creation of the 'contrib' folder and some first contribs. The text is copied from the original mailing list post.

r1507 Adds the "evennia/contrib" folder, a repository of code snippets that are useful for the coder, but optional since they might not be suitable or needed for all types of games. Think of them as building blocks one could use or expand on or have as inspiration for one's own designs.

For me, these primarily help me to test and debug Evennia's API features.

So far, I've added the following optional modules in evennia/contrib:

- Evennia `MenuSystem` - A base set of classes and cmdsets for creating in-game multiple-choice menus in Evennia. The menu tree can be of any depth. Menu options can be numbered or given custom keys, and each option can execute code. Also contains a yes/no question generator function. This is intended to be used by commands and presents a y/n question to the user for accepting an action. Includes a simple new command 'menu' for testing and debugging.
- Evennia Lineeditor - A powerful line-by-line editor for editing text in-game. Mimics the command names of the famous VI text editor. Supports undo/redo, search/replace, regex-searches, buffer formatting, indenting etc. It comes with its own help system. (Makes minute use of the `MenuSystem` module to show a y/n question if quitting without having saved). Includes a basic command '@edit' for activating the editor.
- Talking_NPC - An example of a simple NPC object with which you can strike a menu-driven conversation. Uses the `MenuSystem` to allow conversation options. The npc object defines a command 'talk' for starting the (brief) conversation.

Creating these, I was happy to see that one can really create quite powerful system without any hacking of the server at all - this could all be implemented rather elegantly using normal commands, cmdsets and typeclasses.

I fixed a bunch of bugs and outstanding refactorings. For example, as part of testing out the line-editor, I went back and refurbished the cmdparser - it is now much more straight forward (less bug prone) and supports a much bigger variation of command syntaxes. It's so flexible I even removed the possibility to change its module from settings - it's much easier to simply use command.parse() if you want to customize parsing later down the line. The parser is now also considerably more effective. This is due to an optimization resulting from our use of cmdsets - rather than going

through X number of possible command words and store all combinations for later matching, we now do it the other way around - we merge all cmdsets first, then parse the input looking only for those command names/aliases that we know we have available. This makes for much easier and more effective code. It also means that you can identify commands also if they are missing following whitespace (as long as the match is unique). So the parser would now both understand "look me" as well as "lookme", for example.

Update as of April 2011

This update adds the ability to disconnect from one's puppet and go OOC.

r1484 implements some conceptual changes to the Evennia structure. If you use South, you need to run "manage.py migrate", otherwise you probably have to reset the databases from scratch.

As previously desribed, Evennia impments a strict separation between Player objects (OOC, Out-of-character) objects and Characers (IC In-Character) objects. Players have no existence in the game world, they are abstract representations of connected player sessions. Characters (and all other Objects) have a game-world representation - they can be looked at, they have a location etc. They also used to be the only entities to be able to host cmdsets. This is all well and good as long as you only act as one character - the one that is automatically created for you when you first connect to Evennia. But what if you want to control *another* character (puppet)? This is where the problems start.

Imagine you are an Admin and decide on puppeting a random object. Nothing stops you from doing so, assuming you have the permissions to do so. It's also very easy to change which object you control in Evennia - just switch which object the Player's "character" property points to, and vice-versa for the Objects "player" property (there are safe helper methods for this too). So now you have become the new object. But this object has no commandset defined on it! Not only is now your Admin permissions gone, you can't even get back out, since this object doesn't have a @puppet (or equivalent) command defined for you to use!

On the other hand, it's not a bad idea to be able to switch to an object with "limited" capabilities. If nothing else, this will allow Admins to play the game as a "non-privileged" character if they want - as well as log into objects that have unique commands only suitable for that object (become the huge robot and suddenly have access to the "fire cannon" command sounds sweet, doesn't it?)

Having pondered how to resolve this in a flexible way, Player objects now also has a cmdsethandler and can store cmdsets, the same way as Objects can. Players have a default set of commands defined by settings.CMDSET_OOC. These are applied with a low priority, so same-named commands in the puppeted object will override the ooc command. The most important bit is that commands @ic (same as @puppet) as well as @ooc are now in the OOC command set and always available should you "become" an Object without a cmdset of its own. @ooc will leave your currently controlled character and put you in an "OOC" state where you can't do much more than chat on channels and read help files. @ic will put you back in control of your character again. Admins can @ic to any object on which they pass the "puppet" access lock restriction. You still need to go IC for most of your non-comm administrative tasks, that's the point. For your own game, the ooc state would be a great place for a Character selection/creation screen, for example.

Update as of March 2011

This update introduced the new lock/permission system, replacing an old one where lock and permission where used interchangeably (most confusing). Text was copied from the original mailing list post.

r1346 Adds several revisions to Evennia. Here are a few highlights:

== A revised lock/permission system ==

The previous system combined permissions with locks into one single string called "permissions". While potentially powerful it muddled up what was an access restriction and what was a key. Having a unit "permission" that both dealt with access and limiting also made it very difficult to let anyone but superusers access to change it. The old system also defaulted to giving access, which made for hard-to-detect security holes.

Having pondered this for a while the final straw was when I found that I myself didn't fully understand the system I myself wrote - that can't be a good sign. ^_^;
So, the new system has several changes in philosophy:

- All Evennia entities (commands, objects, scripts, channels etc) have multiple "locks" defined on them. A lock is an "access rule" that limits a certain type of access. There might be one access rule (lock) for "delete", another for "examine" or "edit" but any sort of lock is possible, such as "owner" or "get". No more mix-up between permissions and locks. Permissions should now be read as "keys" and are just one way of many to authenticate.
- Locks are handled by the "locks" handler, such as locks.add(), locks.remove() etc. There is also a convenience function access() that takes the place of the old has_perm() (which is not a fitting name anymore since permissions doesn't work the way they did).
- A lock is defined by a call to a set of lock functions. These are normal python functions that take the involved objects as arguments and establishes if access should be granted or not.
- A system is locked by default. Access is only obtained if a suitable lock grants it.
- All entities now receive a basic set of locks at creation time (otherwise noone besides superuser would have any access)

In practice it works like this:
You try to delete myobject by calling @delete myobject. @delete calls myobject.access(caller, 'delete'). The lockhandler looks up a lock with the access type "delete" and returns a True of False.

Permissions

Only Objects and Players have a "permissions" property anymore, and this is now only used for key strings. A permission has no special standing now - a lock can use any attribute or property to establish access. Permissions do have some nice extra security features out of the box though.

- controlled from @perm, which can be a high-permission command now that locks are separate.
- settings.PERMISSION_HIERARCHY is a tuple of permission strings such as ("Players", "Builders", "Wizards"). The perm() lock function will make sure that higher permissions automatically grants the permissions of those below.

General fixes

As part of testing and debugging the new lock system I fixed a few other issues:

- @reload now asynchonously updates all the objects in the database. This means that you can do nifty things like updating cmdsets on the fly without a server reload!
- Some 30 new unittest cases for commands and locks. Command unittests were refined a lot. This also meant finding plenty of minor bugs in those commands.
- Some inconsistencies in the server/session system had been lingering behind. Fixed now.
- Lots of small fixes.

The wiki is almost fully updated (including the auto-updating command list!), but there might still be text around referring to the old way of doing things. Fix it if you see it. And as usual, report bugs to the issue tracker.

Devel branch as of September 2010

This update added the twisted webserver and webclient. It also moved the default cmdset to src/.

This has been merged into main. The text is copied from the original mailing list post.

Starting with r1245, the underlying server structure of Evennia has changed a bit. The details of protocol implementation should probably mostly be of interest for Evennia developers, but the additions of new web features should be of interest to all.

Maybe the most immediate change you'll notice is that Evennia now defaults to opening two ports, one for telnet and another for a webserver. Yep, Evennia now runs and serves its web presence with its very own Twisted webserver. The webserver, which makes use of Twisted's wsgi features to seamlessly integrate with Django's template system, is found in src/server/webserver.py. The Twisted webserver should be good for most needs. You can of course still use Apache if you really want, but there is now at least no need to use Django's "test server" at all, it all runs by default.

All new protocols should now inherit from src.server.session.Session, a generic class that incoorporate the hooks Evennia use to communicate with all player sessions, such as at_connect(), at_disconnect(), at_data_in(), at_data_out() etc. The all-important msg() function still handles communication from your game to the session, this now also takes an optional keyword 'data' to carry eventual extra parameters that certain protocols might have need for (data is intentionally very vaguely specified, but could for example be instructions from your code for updating a graphical client in some way).

Two protocols are currently written using this new scheme - the standard telnet protocol (now found separately as server/telnet.py) and a web mud client protocol in server/webclient.py.

The web mud client (which requires the web server to be running too) allows for a player to connect to your game through a web browser. You can test it from your newly started game's website. Technically it uses an ajax long polling scheme (sometimes known as 'comet'). The client part running in the browser is a javascript program I wrote using the jQuery javascript library (included in src/web/, although any client and library could be used). The django integration allows for an interesting hybrid, where the Django templating system can be used both for the game website and the client, while the twisted asynchronous reactor handles the real time updating of the client. Please note that the default javascript web client is currently very rough - both it and the underlying protocol still needs work. But it should serve as a hint as to what kind of stuff is possible. The wiki will be updated as the details stabilize.

Unrelated to the new web stuff (but noticeable for game devs) is that the default command set was moved from game/gamesrc/commands/default to src/commands/default since some time. The reason for this change was to make it clearer that these commands are part of the default distribution (i.e. might be updated when you update Evennia) and should thus not be edited by admins - like all things in src/. All this did was to make what was always the best-practice more explicit: To extend the default set, make your own modules in game/gamesrc/commands, or copy them from the default command set. The basecmd.py and basecmdset.py have been updated to clearer explain how to extend things.

Devel branch as of August 2010

This update was a major rewrite of the orginal Evennia, introducing Typeclasses and Scripts as well as Commands, CmdSets and many other features.

Note: The devel branch merged with trunk as of r970 (aug2010). So if you are new to Evennia, this page is of no real interest to you.

Introduction

The Evennia that has been growing in trunk for the last few years is a wonderful piece of software, with which you can do very nice coding work. It has however grown 'organically', adding features here and there by different coders at different times, and some features (such as my State system) were bolted onto an underlying structure for which it was never originally intended.

Meanwhile Evennia is still in an alpha stage and not yet largely used. If one needs to do a cleanup/refactoring and homogenization of the code, now is the time time to do it. So I set out to do just that.

The "devel-branch" of Evennia is a clean rework of Evennia based on trunk. I should point out that the main goal has been to make system names consistent, to add all features in a fully integrated way, and to give all subsystems a more common API for the admin to work against. This means that in the choice between a cleaner implementation and backwards-compatability with trunk, the latter has had to stand back. However, you'll hopefully find that converting old codes shouldn't be too hard. Another goal is to further push Evennia as a full-fledged barebones system for *any* type of mud, not just MUX. So you'll find far more are now user-configurability now than ever before (MUX remains the default though).

Devel is now almost ready for merging with the main trunk, but it needs some more eyes to look at it first. If you are brave and want to help report bugs, you can get it from the *griatch* branch with

```
svn checkout http://evennia.googlecode.com/svn/branches/griatch evennia-devel
```

Concepts changed from trunk to devel

Script parent -> Typeclasses

The biggest change is probably that script parents have been replaced by *typeclasses*. Both handle the abstraction of in-game objects without having to create a separate database model for each (i.e. it allows objects to be anything from players to apples, rooms and swords all with the same django database model).
A script parent in trunk was a class stored in a separate module together with a 'factory' function that the engine called. The admin had to always remember if they were calling a function on the database model or if it in fact sat on the script parent (the call was made through something called the "scriptlink").

By contrast, a typeclass is a normal python class that inherits from the *!TypeClass* parent. There are no other required functions to define. This class uses **getattribute** and **setattr** transparently behind the scenes to store data onto the persistent django object. Also the django model is aware of the typeclass in the reverse direction. The admin don't really have to worry about this connection, they can usually consider the two objects (typeclass and django model) to be one.

So if you have your 'apple' typeclass, accessing, say the 'location', which is stored as a persistent field on the django model, you can now just do `loc = apple.location` without caring where it is stored.

The main drawback with any typeclass/parent system is that it adds an overhead to all calls, and this overhead might be slightly larger with typeclasses than with trunk's script parents although I've not done any testing. You also need to use Evennia's supplied `create` methods to create the objects rather than to create objects with plain Django by instantiating the model class; this so that the rather complex relationships can be instantiated safely behind the scenes.

Command functions + !StateCommands-> Command classes + !CmdSets

In trunk, there was one default group of commands in a list GLOBAL_CMD_TABLE. Every player in game used this. There was a second dictionary GLOBAL_STATE_TABLE that held commands valid only for certain *states* the player might end up in - like entering a dark room, a text editor, or whatever. The problem with this state system, was that it was limited in its use - every player could ever only be in one state at a time for example, never two at the same time. The way the system was set up also explicitly made states something unique to players - an object could not offer different commands dependent on its state, for example.

In devel, *every* command definition is grouped in what's called a *!CmdSet* (this is, like most things in Devel, defined as a class). A command can exist in any number of cmdsets at the same time. Also the 'default' group of commands belong to a cmdset. These command sets are no longer stored globally, but instead locally on each object capable of launching commands. You can add and new cmdsets to an object in a stack-like way. The cmdsets support set operations (Union, Difference etc) and will merge together into one cmdset with a unique set of commands. Removing a cmdset will re-calculate those available commands. This allows you to do things like the following (impossible in trunk):

A player is walking down a corridor. The 'default' cmdset is in play. Now he meets an enemy. The 'combat' cmdset is merged onto (and maybe replacing part of) the default cmdset, giving him new combat-related commands only available during combat. The enemy hits him over the head, dazing him. The "Dazed" cmdset is now added on top of the previous ones - maybe he now can't use certain commands, or might even get a garbled message if trying to use 'look'. After a few moments the dazed state is over, and the 'Dazed' cmdset is removed, returning us to the combat mode we were in before. And so on.

Command definitions used to be functions, but are now classes. Instead of relying on input arguments, all relevant variables are stored directly on the command object at run-time. Also parsing and function execution have been split into two methods that are very suitable for subclassing (an example is all the commands in the default set which inherits from the !MuxCommand class - that's the one knowing about MUX's special syntax with /switches, '=' and so on, Evennia's core don't deal with this at all!).

Example of new command definition:

class CmdTest(Command):

def func(self):

self.caller.msg("This is the test!")

Events + States -> Scripts

The Event system of Evennia used to be a non-persistent affair; python objects that needed to be explicitly called from code when starting. States allowed for mapping different groups of commands to a certain situations (see !CmdSets above for how commands are now always grouped).

Scripts (warning: Not to be confused with the old *script parents*!) are persistent database objects now and are only deleted on a server restart if explicitly marked as non-persistent.

A script can have a time-component, like Events used to have, but it can also work like an 'Action' or a 'State' since a script constantly checks if it is still 'valid' and if not will delete itself. A script handles everything that changes with time in Evennia. For example, all players have a script attached to them that assigns them the default cmdset when logging in.

Oh, and Scripts have typeclasses too, just like Objects, and carries all the same flexibility of the Typeclass system.

User + player -> User + Player + character

In trunk there is no clear separation between the User (which is the django model representing the player connecting to the mud) and the player object. They are both forced to the same dbref and are essentially the same for most purposes. This has its advantages, but the problem is configurability for different game types - the in-game player object becomes the place to store also OOC info, and allowing a player to have many characters is a hassle (although doable, I have coded such a system for trunk privately).

Devel-branch instead separate a "player character" into three tiers:

- The User (Django object)
- The PlayerDB (User profile + Player typeclass)
- The ObjectDB (+ Character typeclass)

User is not something we can get out of without changing Django; this is a permission/password sensitive object through which all Django users connect. It is not configurable to any great extent except through it's *profile*, a django feature that allows you to have a separate model that configures the User. We call this profile 'PlayerDB', and for almost all situations we deal with this rather than User. PlayerDB can hold attributes and is typeclassed just like Objects and Scripts (normally with a typeclass named simply *Player*) allowing very big configurability options (although you can probably get away with just the default setup and use attributes for all but the most exotic designs). The Player is an OOC entity, it is what chats on channels but is not visible in a room.
The last stage is the in-game ObjectDB model, typeclassed with a class called 'Character' by default. This is the in-game object that the player controls.

The neat thing with this separation is that the Player object can easily switch its Character object if desired - the two are just linking to each other through attributes. This makes implementing multi-character game types much easier and less contrived than in the old system.

Help database -> command help + help database

Trunk stores all help entries in the database, including those created dynamically from the command's doc strings. This forced a system where the auto-help creation could be turned off so as to not overwrite later changes made by hand. There was also a mini-language that allowed for creating multiple help entries from the `__doc__` string.

Devel-branch is simpler in this regard. All commands are *always* using `__doc__` on the fly at run time without hitting the database (this makes use of cmdsets to only show help for commands actually available to you). The help database is stand-alone and you can add entries to it as you like, the help command will look through both sources of help entries to match your query.

django-perms + locks -> permission/locks

Trunk relies on Django's user-permissions. These are powerful but have the disadvantage of being 'app-centric' in a way that makes sense for a web app, not so much for a mud.
The devel-branch thus implements a completely stand-alone permission system that incoorperate both permissions and locks into one go - the system uses a mini-language that has a permission string work as a keystring in one situation and as a complex lock (calling python lock functions you can define yourself) in another.

The permission system is working on a fundamental level, but the default setup probably needs some refinements still.

Mux-like comms -> Generic comms

The trunk comm system is decidedly MUX-like. This is fine, but the problem is that much of that mux-likeness is hard-coded in the engine.

Devel just defines three objects, Channel and Msg and an object to track connections between players and channels (this is needed to easily delete/break connections). How they interact with each other is up to the commands that use them, making the system completely configurable by the admin.

All ooc messages - to channels or to players or both at the same time, are sent through use of the Msg object. This means a full log of all communications become possible to keep. Other uses could be an e-mail like in/out box for every player. The default setup is still mux-like though.

Hard-coded parsing -> user customized parsing

Essentially all parts of parsing a command from the command line can be customized. The main parser can be replaced, as well as error messages for multiple-search matches.
There is also a considerable difference in handling exits and channels - they are handled as commands with their separate cmdsets and searched with the same mechanisms as any command (almost any, anyway).

Aliases -> Nicks

Aliases (that is, you choosing to for yourself rename something without actually changing the object itself) used to be a separate database table. It is now a dictionary 'nicks' on the Character object - that replace input commands, object names and channel names on the fly. And due to the separation between Player and Character, it means each character can have its own aliases (making this a suitable start for a recog system too, coincidentally).

Attributes -> properties

To store data persistently in trunk requires you to call the methods `get_attribute_value(attr)` and `set_attribute(attr, value)`. This is available for in-game Objects only (which is really the only data type that makes sense anyway in Trunk).

Devel allows attribute storage on both Objects, Scripts and Player objects. The attribute system works the same but now offers the option of using the `db` (for database) directly. So in devel you could now just do:
obj.db.attr = value
value = obj.db.attr
And for storing something non-persistently (stored only until the server reboots) you can just do
obj.attr = value
value = obj.attr
The last example may sound trivial, but it's actually impossible to do in trunk since django objects are not guaranteed to remain the same between calls (only stuff stored to the database is guaranteed to remain). Devel makes use of the third-party `idmapper` functionality to offer this functionality. This used to be a very confusing thing to new Evennia admins.

All database fields in Devel are now accessed through properties that handle in/out data storage. There is no need to save() explicitly anymore; indeed you should ideally not need to know the actual Field names.

Always full persistence -> Semi/Full persistence

In Evennia trunk, everything has to be saved back/from the database at all times, also if you just need a temporary storage that you'll use only once, one second from now. This enforced full persistency is a good thing for most cases - especially for web-integration, where you want the world to be consistent regardless of from where you are accessing it.

Devel offer the ability to yourself decide this; since semi-persistent variables can be stored on objects (see previous section). What actually happens is that such variables are stored on a normal python object called `ndb` (non-database), which is transparently accessed. This does not touch the database at all.

Evennia-devel offers a setting `FULL_PERSISTENCE` that switches how the server operates. With this off, you have to explicitly assign attributes to database storage with e.g. `obj.db.attr = value`, whereas normal assignment (`obj.attr = value`) will be stored non-persistent. With `FULL_PERSISTENT` on however, the roles are reversed. Doing `obj.attr = value` will now actually be saving to database, and you have to explicitly do `obj.ndb.attr = value` if you want non-persistence. In the end it's a matter of taste and of what kind of game/features you are implementing. Default is to use full persistence (but all of the engine explicitly put out `db` and `ndb` making it work the same with both).

Commonly used functions/concept that changed names

There used to be that sending data to a player object used a method `emit_to()`, whereas sending data to a session used a method `msg()`. Both are now called `msg()`. Since there are situations where it might be unclear if you receive a session or a player object (especially during login/logout), you can now use simply use `msg()` without having to check (however, you *can* still use `emit_to` for legacy code, it's an alias to msg() now). Same is true with emit_to_contents() -> msg_to_contents().

`source_object` in default commands are now consistently named *caller* instead.

`obj.get_attribute_value(attr)` is now just `obj.get_attribute(attr)` (but see the section on Attributes above, you should just use `obj.db.attr` to access your attribute).

How hard is it to convert from trunk to devel?

It depends. Any game logic game modules you have written (AI codes, whatever) should ideally not do much more than take input/output from evennia. These can usually be used straight off.

Commands and Script parents take more work but translate over quite cleanly since the idea is the same.
For commands, you need to make the function into a class and add the parse(self) and func(self) methods (parse should be moved into a parent class so you don't have to use as much double code), as well as learn what variable names is made available (see the commands in `gamesrc/commands/default` for guidance). You can make States into !CmdSets very easy - just listing the commands needed for the state in a new !CmdSet.

Script parents are made into Typeclasses by deleting the factory function and making them inherit from a !TypeClassed object (such as Object or Player) like the ones in `gamesrc/typeclasses/basetypes.py`, and then removing all code explicitly dealing with script parents.

Converting to the new Scripts (again, don't confuse with the old *script parents*!) is probably the trickiest, since they are a more powerful incarnation of what used to be two separate things; States and Events. See the examples in the `gamesrc/scripts/` for some ideas.

Better docs on all of this will be forthcoming.

Things not working/not implemented in devel (Aug 2010)

All features planned to go into Devel are finished. There are a few
features available in Trunk that is not going to work in Devel until after

it merges with Trunk:

- IMC2/IRC support is not implemented.
- Attribute-level permissions are not formalized in the default cmdset.
- Some of the more esoteric commands are not converted.

Please play with it and report bugs to our bug tracker!