shellbot Documentation

Release 17.8.6

Bernard Paques

Sep 20, 2017

Contents

CHAPTER 1

Shellbot: Python Chat Bot Framework

Fast, simple and lightweight micro bot framework for Python. It is distributed as a single package and has very few dependencies other than the Python Standard Library. Shellbot supports Python 3 and Python 2.7. Test coverage exceeds 90%.

- **Channels:** a single bot can access jointly group and direct channels
- **Commands:** routing from chat box to function calls made easy, including support of file uploads
- **State machines:** powerful and pythonic way to bring intelligence to your bot
- **Stores:** each bot has a dedicated data store
- **Utilities:** convenient configuration-driven approach, chat audit, and more
- **Platforms:** Cisco Spark, local disconnected mode for tests – looking for more

The Batman example

```
import os
import time

from shellbot import Engine, Context, Command
Context.set_logger()


class Batman(Command):  # a command that displays static text
    keyword = 'whoareyou'
    information_message = u"I'm Batman!"


class Batcave(Command):  # a command that reflects input from the end user
    keyword = 'cave'
    information_message = u"The Batcave is silent..."
```

```
    def execute(self, bot, arguments=None, **kwargs):
        if arguments:
            bot.say(u"The Batcave echoes, '{0}'".format(arguments))
        else:
            bot.say(self.information_message)


class Batsignal(Command):  # a command that uploads a file/link
    keyword = 'signal'
    information_message = u"NANA NANA NANA NANA"
    information_file = "https://upload.wikimedia.org/wikipedia/en/c/c6/Bat-signal_
↪1989_film.jpg"

    def execute(self, bot, arguments=None, **kwargs):
        bot.say(self.information_message,
                file=self.information_file)


class Batsuicide(Command):  # a command only for group channels
    keyword = 'suicide'
    information_message = u"Go back to Hell"
    in_direct = False

    def execute(self, bot, arguments=None, **kwargs):
        bot.say(self.information_message)
        bot.dispose()


engine = Engine(  # use Cisco Spark and load shell commands
    type='spark',
    commands=[Batman(), Batcave(), Batsignal(), Batsuicide()])

os.environ['BOT_ON_ENTER'] = 'You can now chat with Batman'
os.environ['BOT_ON_EXIT'] = 'Batman is now quitting the room, bye'
os.environ['CHAT_ROOM_TITLE'] = 'Chat with Batman'
engine.configure()  # ensure that all components are ready

engine.bond(reset=True)  # create a group channel for this example
engine.run()  # until Ctl-C
engine.dispose()  # delete the initial group channel
```

Quick installation

To install the shellbot package, type:

```
$ pip install shellbot
```

Or, if you prefer to download the full project including examples and documentation, and install it, do the following:

```
$ git clone https://github.com/bernard357/shellbot.git
$ cd shellbot
$ pip install -e .
```

Where do you want to start?

- Documentation: Shellbot at ReadTheDocs
- Python package: Shellbot at PyPi
- Source code: Shellbot at GitHub
- Free software: Apache License (2.0)

Credits

- securitybot from the Dropbox team
- Bottle
- ciscosparkapi
- PyYAML
- Cookiecutter
- cookiecutter-pypackage

CHAPTER 2

Install the shellbot package

Shellbot is available as a python package, so the installation, the upgrade, and the removal of the software are really easy.

Install the shellbot package

Shellbot is available on PyPi. You can install latest stable version using pip.

On Linux or on Mac OSX:

```
$ sudo apt-get install python-pip
$ sudo pip install shellbot
```

On Windows:

```
> pip install virtualenv
> virtualenv.exe .
> .\Script\pip install shellbot
```

Upgrade the shellbot package

If you have used pip to install the software, then you can use it again to upgrade the package:

```
$ sudo pip install --upgrade shellbot
```

Remove the shellbot package

Why would you bother about a small set of files at a computer? Anyway, if needed here is the command to remove Shellbot from a python environment:

```
$ sudo pip uninstall shellbot
```

CHAPTER 3

Useful examples of Shellbot use cases

Each example below is provided as a complete software that you can run by yourself. This list is roughly ordered in growing complexity so if you are new to shellbot it is recommended that you start by the beginning.

Hello, World!

Is this the most long-lasting contribution of Kernighan and Ritchie in their famous book on the C language? Anyway, here we go with a quick start on shellbot.

View the source code of this example

How to execute a command?

Within shellbot, a command is simply a Python object with a member function `execute()` and an attribute `keyword`. Maybe with a bare example this will become much clearer:

```
class Hello(Command):
    keyword = 'hello'

    def execute(self, bot, **kwargs):
        bot.say(u"Hello, World")
```

And that's it. if you pass an instance of `Hello` to the engine, it will be invoked each time you send `hello` to the bot in the chat.

Got it. Can you provide a bit more?

Ok, here is the actual code featured in the Hello World example:

```
class Hello(Command):
    keyword = 'hello'
    information_message = u"Hello, World!"

    feedback_content = u"Hello, **{}**!"
    thanks_content = u"Thanks for the upload of `{}`"

    def execute(self, bot, arguments=None, attachment=None, url=None, **kwargs):

        bot.say(content=self.feedback_content.format(
            arguments if arguments else 'World'))

        if attachment:
            bot.say(content=self.thanks_content.format(attachment))
```

The signature of the `execute()` function show additional arguments for commands that manage file uploads. Also, we provide to `say()` content that is formatted in Markdown so that the rendering in chat is improved.

How to feed shellbot with commands?

By itself, shellbot provides only the `help` command. The command `hello` can be added to the engine during initialization:

```
engine = Engine(command=Hello(), ...)
```

Of, course, a full set of commands can be provided to the engine. In the *Chat with Batman* example, we do:

```
engine = Engine(commands=[Batman(), Batcave(), Batsignal(), Batsuicide()], ...)
```

How to change the banner?

The banner is sent to the chat area when the bot joins a new channel. Shellbot support bare text, rich content, and even file uploads, altogether. This can be changed by adjusting some environment variables, as shown below:

```
os.environ['BOT_BANNER_TEXT'] = u"Type '@{} help' for more information"
os.environ['BOT_BANNER_CONTENT'] = (u"Hello there! "
                                    u"Type ``@{} help`` at any time and  get "
                                    u"more information on available commands.")
os.environ['BOT_BANNER_FILE'] = \
    "http://skinali.com.ua/img/gallery/19/thumbs/thumb_m_s_7369.jpg"

engine.configure()
```

In this example environment variables are set within the python code itself, yet for a regular application this should be done in a separate configuration file.

How to select a chat platform?

The chat platform is selected during the initialization of the engine. Here we put `type='spark'` to select Cisco Spark and that's it:

```
engine = Engine(type='spark', command=Hello())
```

Ok, in addition to this code you also have to set some variables to make it work, but this is regular configuration, done outside the code itself.

Does this manage multiple channels?

Shellbot powers as many channels as necessary from a single engine. In this example a sample channel is created, yet you can invite the bot to any number of other channels, or to your direct channel as well. Here you go:

```
engine.bond(reset=True)  # create a group channel for this example
engine.run()  # until Ctl-C
engine.dispose()  # delete the initial group channel
```

Commands: hello, help

The `hello` command is coming from this example itself, while `help` is built in shellbot.

hello response: Hello, World!

hello Machine response: Hello, Machine!

help response:

```
Available commands:
hello - Hello, World!
help - Show commands and usage
```

help hello response:

```
hello - Hello, World!
usage: hello
```

help help response:

```
help - Show commands and usage
usage: help <command>
```

How to run this example?

To run this script you have to provide a custom configuration, or set environment variables instead:

- `CHANNEL_DEFAULT_PARTICIPANTS` - Mention at least your e-mail address
- `CISCO_SPARK_BOT_TOKEN` - Received from Cisco Spark on bot registration
- `SERVER_URL` - Public link used by Cisco Spark to reach your server

The token is specific to your run-time, please visit Cisco Spark for Developers to get more details:

https://developer.ciscospark.com/

For example, if you run this script under Linux or macOs with support from ngrok for exposing services to the Internet:

```
export CHANNEL_DEFAULT_PARTICIPANTS="alice@acme.com"
export CISCO_SPARK_BOT_TOKEN="<token id from Cisco Spark for Developers>"
export SERVER_URL="http://1a107f21.ngrok.io"
python hello.py
```

Chat with Batman

In this example the bot pretends to be Batman, and supports some commands that are making sense in this context.

View the source code of this example

How to build a dynamic response?

Look at the command `cave`, where the message pushed to the chat channel depends on the input received. This is done with regular python code in the member function `execute()`:

```
class Batcave(Command):
    keyword = 'cave'
    information_message = u"The Batcave is silent..."

    def execute(self, bot, arguments=None, **kwargs):
        if arguments:
            bot.say(u"The Batcave echoes, '{0}'".format(arguments))
        else:
            bot.say(self.information_message)
```

Of course, for your own application, it is likely that tests would be a bit more complicated. For example, you could check data from the bot store with `bot.recall()`, or specific settings of the engine with `bot.engine.get()`, or use a member attribute of the command itself. This is demonstrated in other examples.

How to upload files?

The command `signal` demonstrates how to attach a link or a file to a message:

```
class Batsignal(Command):
    keyword = 'signal'
    information_message = u"NANA NANA NANA NANA"
    information_file = "https://upload.wikimedia.org/wikipedia/en/c/c6/Bat-signal_
↪1989_film.jpg"

    def execute(self, bot, arguments=None, **kwargs):
        bot.say(self.information_message,
                file=self.information_file)
```

Here we use some public image, yet the same would work for the upload of a local file:

```
bot.say('my report', file='./shared/reports/August-2017.xls')
```

In a nutshell: with shellbot, files are transmitted along the regular function `say()`.

What about commands that do not apply to direct channels?

When you have this requirement, set the command attribute `in_direct` to `False`. In this example, the bot is not entitled to delete a private channel. So we disable the command `suicide` from direct channels:

```
class Batsuicide(Command):
    keyword = 'suicide'
    information_message = u"Go back to Hell"
    in_direct = False
```

```
    def execute(self, bot, arguments=None, **kwargs):
        bot.say(self.information_message)
        bot.dispose()
```

If you use the command `help` both in group channel and in direct channel, you will see that the list of available commands is different.

How to load multiple commands?

Since each command is a separate object, you can add them as a list bundle to the engine:

```
engine = Engine(
    type='spark',
    commands=[Batman(), Batcave(), Batsignal(), Batsuicide()])
```

In this example we create an instance from each class, and put that in a list for the engine.

BatCommands: whoareyou, cave, signal, suicide

These are a bit more sophisticated than for the *Hello, World!* example, but not much.

whoareyou response: I'm Batman!

cave response: The Batcave is silent...

cave give me some echo response: The Batcave echoes, 'give me some echo'

signal response: NANA NANA NANA NANA

This command also uploads an image to the chat channel.

suicide response: Going back to Hell

The command also deletes the channel where it was executed. It is available only within group channels, and not in direct channels.

How to run this example?

To run this script you have to provide a custom configuration, or set environment variables instead:

- `CHANNEL_DEFAULT_PARTICIPANTS` - Mention at least your e-mail address
- `CISCO_SPARK_BOT_TOKEN` - Received from Cisco Spark on bot registration
- `SERVER_URL` - Public link used by Cisco Spark to reach your server

The token is specific to your run-time, please visit Cisco Spark for Developers to get more details:

https://developer.ciscospark.com/

For example, if you run this script under Linux or macOs with support from ngrok for exposing services to the Internet:

```
export CHANNEL_DEFAULT_PARTICIPANTS="alice@acme.com"
export CISCO_SPARK_BOT_TOKEN="<token id from Cisco Spark for Developers>"
export SERVER_URL="http://1a107f21.ngrok.io"
python batman.py
```

Credit: https://developer.ciscospark.com/blog/blog-details-8110.html

Notify

In this example we use the bot only for easy notifications to a space. There is no command in the shell at all, and the bot is not even started.

View the source code of this example

How to create a bot and configure it in one line?

The simplest approach is to set environment variables and then to create the bot. This can be done externally, before running the program, for secret variables such as the Cisco Spark token (see below). Or variables can be set directly from within the script itself, as `CHAT_ROOM_TITLE` in this example.

How to create or to delete a channel?

When you access a bot for the first time it is created automatically in the back-end platform. From a software perspective, call `engine.get_bot()` and this will give you a bot instance.

The bot itself can be used when you have to delete a channel, with a call of `bot.dispose()`.

How to post a notification?

Use `bot.say()` on the bot instance. Messages posted can feature bare or rich text, and you can also upload an image or a document file.

Why do we not start the bot?

There is no call to `bot.run()` here because there is no need for an active shell. The program updates a channel, however is not interactive and cannot answer messages send to it. Of course, it is easy to implement a couple of commands at some point so that you evolve towards a responsive bot.

How to run this example?

To run this script you have to provide a custom configuration, or set environment variables instead:

- `CHANNEL_DEFAULT_PARTICIPANTS` - Mention at least your e-mail address
- `CISCO_SPARK_BOT_TOKEN` - Received from Cisco Spark on bot registration
- `SERVER_URL` - Public link used by Cisco Spark to reach your server

The token is specific to your run-time, please visit Cisco Spark for Developers to get more details:

https://developer.ciscospark.com/

For example, if you run this script under Linux or macOs with support from ngrok for exposing services to the Internet:

```
export CHANNEL_DEFAULT_PARTICIPANTS="alice@acme.com"
export CISCO_SPARK_BOT_TOKEN="<token id from Cisco Spark for Developers>"
export SERVER_URL="http://1a107f21.ngrok.io"
python notify.py
```

Fly with Buzz – "To infinity and beyond"

In this example we deal with commands that take significant time to execute. How to run long-lasting transactions in the background, so that the bot stays responsive?

View the source code of this example

Buzz is flying from Earth to some planets and come back. Obviously, this is the kind of activity that can take ages, yet here each mission lasts about 30 seconds.

Ok. So, when I type `explore Uranus` in the chat box, do I have to wait for 30 seconds before the next command is considered? Hopefully not!

How to execute commands asynchronously?

The two commands `explore` and `blast` are non-interactive. This means that they are pushed to a pipeline for background execution. With this concept, you can get a dialog similar to the following:

```
> buzz explore Mercury
Ok, I am working on it
#1 - Departing to Mercury
> buzz blast Neptune
Ok, will work on it as soon as possible
#1 - Approaching Mercury
#1 - Landed on Mercury
> buzz planets
Available destinations:
- Venus
- Moon

...
```

In other terms, the bot is always responsive, whatever is executing in the background. Also, non-interactive commands are executed in the exact sequence of their submission.

These concepts are implemented with instances of `Rocket` that are attached to bots (Rocket source code). Every rocket has a queue that receives commands submitted in the chat box. And of course, every rocket is running a separate process to pick up new missions and to execute them.

How to attach a rocket and make it fly, for every bot?

Since the objective is that each bot has its own rocket attached, we provide with a custom driver that does exactly this:

```
class FlyingBot(ShellBot):  # add a rocket to each bot
    def on_init(self):
        self.rocket = Rocket(self)
        self.rocket.start()
```

Then the engine is instructed to use this custom driver instead of the regular one:

```
engine = Engine(driver=FlyingBot, ...)
```

With this way of working, each time the bot is invited to a channel (direct or group), a new rocket is instantiated and ready to go.

How can a command interact with the rocket?

The command delegates a new mission with a simple function call, like for example in the command `explore`:

```
class Explore(Command):
    keyword = u'explore'
    information_message = u'Explore a planet and come back'
    usage_message = u'explore <destination>'

    def execute(self, bot, arguments=None, **kwargs):
        """
        Explores a planet and comes back
        """

        if not arguments:
            bot.say(u"usage: {}".format(self.usage_message))
            return

        bot.rocket.go('explore', arguments)
```

On rocket side, the mission is pushed to a queue for later processing:

```
def go(self, action, planet):
    """Engages a new mission"""

    self.inbox.put((action, planet))
```

Within the rocket instance, a process is continuously monitoring the `inbox` queue to pick up new missions and to execute them, one at a time.

How to store data separately for each bot?

With shellbot, each bot is coming with its own data store, that is distinct from data stores of other bots. Content of the bot store can be statically initialized by the engine itself, if settings starting with the label `bot.store` are provided. This mechanism is used in this example for listing available planets:

```
engine.set(
    'bot.store.planets',
    ['Mercury', 'Venus', 'Moon', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'])
```

The function `bot.recall()` can then be used to retrieve the list of planets. This is exactly what is done for the command `planets`:

```
class Planets(Command):
    keyword = u'planets'
    information_message = u'List reachable planets'

    list_header = u"Available destinations:"

    def execute(self, bot, arguments=None, **kwargs):
        """
        Displays the list of available planets
        """

        items = bot.recall('planets', [])
        if len(items):
            bot.say(self.list_header
```

```
                    + '\n- ' + '\n- '.join(items))
        else:
            bot.say(u"Nowhere to go right now.")
```

When a planet has been blasted, it is removed from the data store with code similar to this:

```
items = self.bot.recall('planets', [])
items.remove(planet)
self.bot.remember('planets', items)
```

Keep in mind that the list of available planets evolve over time, since some of them can be nuked by end users. So, if Mercury is blasted in one channel, and Neptune in another channel, there is a need for independent management of planets across bots. This is exactly what `bot.remember()` and `bot.recall()` provide, hopefully.

Commands: planets, explore, blast

planets provides a list of available destinations

explore <planet> you then track in real-time the progress of the mission

blast <planet> similar to exploration, except that the planet is nuked

How to run this example?

To run this script you have to provide a custom configuration, or set environment variables instead:

- `CHANNEL_DEFAULT_PARTICIPANTS` - Mention at least your e-mail address
- `CISCO_SPARK_BOT_TOKEN` - Received from Cisco Spark on bot registration
- `SERVER_URL` - Public link used by Cisco Spark to reach your server

The token is specific to your run-time, please visit Cisco Spark for Developers to get more details:

https://developer.ciscospark.com/

For example, if you run this script under Linux or macOs with support from ngrok for exposing services to the Internet:

```
export CHANNEL_DEFAULT_PARTICIPANTS="alice@acme.com"
export CISCO_SPARK_BOT_TOKEN="<token id from Cisco Spark for Developers>"
export SERVER_URL="http://1a107f21.ngrok.io"
python buzz.py
```

Hotel California

In this example we show how to keep people in the same channel.

View the source code of this example

How to preserve state of the hotel?

In this example a bot is representing a distinct hotel instance. So there is a need to know, for each bot, is the hotel is retaining visitors or not.

This is done with `bot.remember()` and `bot.recall()` calls respectively.

Under the hood, each bot is equipped with a dedicated data store.

How to know that someone is joining or leaving?

Shellbot implements a simple event dispatcher, that is used in this example to detect when people join or leave a channel.

For this, we create a python object with functions `on_join()` and `on_leave()`. Then this handler is registered for the events `join` and `leave` generated by the engine.

How to add participants to a channel?

You can add a person to a channel by invoking `bot.add_participant()` with the e-mail address of a new participant. Note that some restrictions may apply, depending on the commercial agreement with the cloud service provider.

Commands: open, close, hotel

The commands prevent or allow people to leave the Hotel, or report on current status.

open This command puts the bot in "sticky" mode. If a participant leaves a channel for some reason, he is automatically added back after 5 seconds.

close After this command, participants can leave the channel freely. They are not forced to come back anymore

hotel This command displays the current mode of working. Can participants go away or not?

How to run this example?

To run this script you have to provide a custom configuration, or set environment variables instead:

- `CHANNEL_DEFAULT_PARTICIPANTS` - Mention at least your e-mail address
- `CISCO_SPARK_BOT_TOKEN` - Received from Cisco Spark on bot registration
- `SERVER_URL` - Public link used by Cisco Spark to reach your server

The token is specific to your run-time, please visit Cisco Spark for Developers to get more details:

https://developer.ciscospark.com/

For example, if you run this script under Linux or macOs with support from ngrok for exposing services to the Internet:

```
export CHANNEL_DEFAULT_PARTICIPANTS="alice@acme.com"
export CISCO_SPARK_BOT_TOKEN="<token id from Cisco Spark for Developers>"
export SERVER_URL="http://1a107f21.ngrok.io"
python hotel_california.py
```

CHAPTER 4

How to contribute to Shellbot?

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

> "80% of success is just showing up." - Woody Allen

Contributing to open source for the first time can be scary and a little overwhelming. This project is like the others. It needs help and that help can mean using it, sharing the information, supporting people, whatever. The thing that folks forget about open source is that it's most volunteers who are doing it for the love of it. They show up.

Contents

You are not a developer? We are glad that you are involved.

We want you to feel as comfortable as possible with this project, whatever your skills are. Here are some ways to contribute:

- use it for yourself
- communicate about the project
- submit feedback
- report a bug
- write or fix documentation
- fix a bug or an issue
- implement some feature

How to use shellbot for yourself?

Initially the shellbot project has been initiated so that as many IT professionals as possible can develop bots in python. Look at examples coming with the framework and see how they can accelerate your own projects. Duplicate one example for yourself, expand it and enjoy!

How to communicate about the shellbot project?

If you believe that the project can help other persons, for whatever reason, just be social about it. Use Facebook, LinkedIn, Twitter, or any other tool that are used by people you dare. Lead them to build a bot with shellbot.

How to submit feedback?

The best way to send feedback, positive or negative, is to file an issue. At first sight it may seems strange to mix feedback with issues. In practice this is working smoothly because there is a single place for asynchronous interactions across the project community.

Provide feedback right now

Your use case may be new, and therefore interesting to us. Or you may raise the hand and explain your own user experience, bad or good. Ask a question if something is not clear. If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

How to report a bug?

Have you identified some bug? This is great! There is a single place for all bugs related to this project:

Shellbot bugs and issues

This is where issues are documented and discussed before they are fixed by the community. Each time you report a bug, please include:

- Your operating system name and version.

- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

How to improve the documentation?

The project could always use more documentation, for sure. Currently documentation is generated automatically from GitHub updates and made available at ReadTheDocs.

Shellbot primary documentation

Documentation files in reST format (i.e., text files with an extension `.rst`) sit in the `docs` directory. Images are put in `docs\_static`. Therefore, updating the documentation is as simple as changing any project source file.

So if you already used a text editor, and made some screenshots, please consider to improve project documentation.

For example, here are the typical steps required for the addition of a new tutorial page:

1. From the project page at GitHub, you can fork it so that you have your own project space. If you do not have a GitHub account, please create one. This is provided for free, and will make you a proud member of a global community that matters.
2. Clone the forked project to your workstation so that you get a copy of all files there. GitHub Desktop is recommended across all platforms it supports.
3. Open a text editor and write some text in reST format. Then save it to a new document, e.g., `docs\tutorial01.rst`
4. Make some screen shots or select some pictures that will be displayed within the document. Put all of them in `docs\_static`.
5. Commit changes and push them back to GitHub. On GitHub Desktop this is as simple as clicking on the Sync button.
6. Visit the forked project page at GitHub and navigate to the new documentation page. The reST will be turned automatically to a web page, so that you can check everything. Go back to step 4 and iterate as much as needed.
7. When you are really satisfied by your work, then submit it to the community with a Pull Request. Again, in GitHub Desktop this is a simple click on a button.
8. All Pull Requests are listed from the original project page so you can monitor what the community is doing with them, and jump in anytime.
9. When your Pull Request is integrated, then your contribution is becoming an integral part of the project, and you become an official contributor. Thank you!

How to fix a bug?

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

How to implement new features?

Look through the GitHub issues for features. Anything tagged with "enhancement" is open to whoever wants to implement it.

Ready to contribute? Here's how to set up Shellbot for local development

1. Fork the *shellbot* repo on GitHub. If you do not have an account there yet, you have to create one, really. This is provided for free, and will make you a proud member of a global community that matters. Once you have authenticated, visit the Shellbot repository at GitHub and click on the *Fork* link.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/shellbot.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv shellbot
$ cd shellbot/
$ pip install -e .
$ pip install -r requirements_test.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests:

```
$ make lint
$ make test
$ make coverage
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Some guidelines for your next Pull Request

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in `README.rst`.
3. Check Shellbot continuous integration at Travis CI and make sure that the tests pass there.

CHAPTER 5

Frequently asked questions

This page is updated with questions frequently asked to the team.

Contents

About project governance

Where is this project coming from?

The shellbot project started as an initiative from some colleagues within Dimension Data. We saw a need for a simple, rock-solid and fast package that would allow us to support upcoming client projects.

Is this software available to anyone?

Yes. The software and the documentation have been open-sourced from the outset, so that it can be useful to the global community of bot practioners. The shellbot project is based on the Apache License.

Do you accept contributions to this project?

Yes. There are multiple ways for end-users and for non-developers to contribute to this project. For example, if you hit an issue, please report it at GitHub. This is where we track issues and report on corrective actions. More information at *How to contribute to Shellbot?*

And if you know how to clone a GitHub project, we are happy to consider pull requests with your modifications. This is the best approach to submit additional examples, or updates of the documentation, or evolutions of the python code itself.

How do you structure version numbering?

We put the year, month, and date of the release. So for example version 17.5.7 has been generated on May-5th, 2017.

About shellbot design

What is needed to run shellbot?

Shellbot requires essentially a recent python interpreter. Both versions 2.x and 3.x are accepted. We are aiming to limit dependencies to the very minimum, and to leverage the standard python library as much as possible. Of course, we use some external library related to Cisco Spark. For development and tests, even a small computer can be used, providing that it is connected to the Internet. This can be your own workstation or even a small computer like a Raspberry Pi. Or any general-purpose computer, really. And, of course, it can be a virtual server running in the cloud.

What systems are compatible with shellbot?

Currently, shellbot can interact with Cisco Spark. Our mid-term objective is that it can interface with multiple systems. The architecture is open, so that it can be extended quite easily. We are looking for the addition of Slack, Gitter and Microsoft Teams. If you are interested, or have other ideas, please have a look at *How to contribute to Shellbot?*

About shellbot deployment

How to install shellbot?

Shellbot uses the standard approach for the distribution of python packages via PyPi. In other terms:

```
pip install shellbot
```

Check *Install the shellbot package* for more details.

Is it required to know python?

Yes. Shellbot is a powerful framework with a simple interface. You can start small and expand progressively.

How to run shellbot interactively?

Shellbot is a framework, not a end product on its own. If you get a local copy of the project from GitHub, then go to directory `examples` and run direclty one of the python script there.

For example to run the Buzz example:

```
$ python buzz.py
```

Break the infinite pumping loop if needed with the keystroke *Ctrl-X*.

My question has not been addressed here. Where to find more support?

Please raise an issue at the GitHub project page and get support from the project team.

CHAPTER 6

shellbot package

Subpackages

shellbot.commands package

Submodules

shellbot.commands.audit module

class shellbot.commands.audit.**Audit**(*engine=None*, ***kwargs*)

Bases: *shellbot.commands.base.Command*

Checks and changes audit status

In essence, audit starts with the capture of information in real-time, and continues with the replication of information.

A typical use case is the monitoring of interactions happening in a channel, for security reasons, for compliancy or, simply speaking, for traceability.

Audit can be suspended explicitly by channel participants. This allows for some private exchanges that are not audited at all. However, then command is put in the audit log itself, so that people can be queried afterwards on their private interactions. If the parameter `off_duration` is set, then it is used by a watchdog to restart auditing. Else it is up to channel participants to activate or to de-activate auditing, at will.

The command itself allows for suspending or restarting the audit process. When audit has been activated in a channel, the attribute `audit.switch.<channel_id>` is set to `on` in the context. This can be checked by the observer while handling inbound records.

The audit has to be armed beforehand, and this is checked from the context attribute `audit.has_been_armed`. In normal cases, audit is armed from the underlying space by setting this attribute to True.

`already_off_message` = u'Chat interactions are already private.'

already_on_message = u'Chat interactions are already audited.'

audit_off(*bot*)
Activates private mode

Parameters **bot** (*Shellbot*) – The bot for this execution

audit_on(*bot*)
Activates audit mode

Parameters **bot** (*Shellbot*) – The bot for this execution

audit_status(*bot*)
Reports on audit status

Parameters **bot** (*Shellbot*) – The bot for this execution

disabled_message = u'Audit has not been enabled.'

execute(*bot*, *arguments=None*, ***kwargs*)
Checks and changes audit status

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (*str*) – either 'on' or 'off'

has_been_enabled
Are we ready for auditing or not?

Return type bool

in_direct = False

information_message = u'Check and change audit status'

keyword = u'audit'

off_duration = 60

off_message = u'Chat interactions are not audited.'

on_bond(*bot*)
Activates audit when a bot joins a channel

on_init()
Registers callback from bot

on_message = u'Chat interactions are currently audited.'

on_off(*bot*)
Triggers watchdog when audit is disabled

temporary_off_message = u'Please note that auditing will restart after {}'

usage_message = u'audit [on|off]'

watchdog(*bot*)
Ensures that audit is restarted

shellbot.commands.base module

class shellbot.commands.base.**Command**(*engine=None*, ***kwargs*)
Bases: object

Implements one command

execute (*bot*, *arguments=None*, ***kwargs*)

Executes this command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

The function is invoked with a variable number of arguments. Therefore the need for `**kwargs`, so that your code is safe in all cases.

The recommended signature for commands that handle textual arguments is the following:

‘‘‘ def execute(self, bot, arguments=None, **kwargs):

... if arguments:

...

‘‘‘

In this situation, `arguments` contains all text typed after the verb itself. For example, when the command `magic` is invoked with the string:

```
magic rub the lamp
```

then the related command instance is called like this:

```
magic = shell.command('magic')
magic.execute(bot, arguments='rub the lamp')
```

For commands that can handle file attachments, you could use following approach:

```
def execute(self,
            bot,
            arguments=None,
            attachment=None,
            url=None,
            **kwargs):
    ...
    if url:  # a document has been uploaded with this command
        content = bot.space.download_attachment(url)
        ...
```

Reference information on parameters provided by the shell:

- `bot` - This is the bot instance for which the command is executed. From this you can update the chat with `bot.say()`, or access data attached to the bot in `bot.store`. The engine and all global items can be access with `bot.engine`.
- `arguments` - This is a string that contains everything after the command verb. When `hello How are you doing?` is submitted to the shell, `hello` is the verb, and `How are you doing?` are the arguments. This is the regular case. If there is no command `hello` then the command `*default` is used instead, and arguments provided are the full line `hello How are you doing?`.
- `attachment` - When a file has been uploaded, this attribute provides its external name, e.g., `picture024.png`. This can be used in the executed command, if you keep in mind that the same name can be used multiple times in a conversation.

- `url` - When a file has been uploaded, this is the handle by which actual content can be retrieved. Usually, ask the underlying space to get a local copy of the document.

This function should report on progress by sending messages with one or multiple `bot.say("Whatever response")`.

in_direct = True

in_group = True

information_message = None

is_hidden = False

keyword = None

on_init()

Handles extended initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self):
    self.engine.register('stop', self)
```

usage_message = None

shellbot.commands.close module

class shellbot.commands.close.**Close**(*engine=None*, ***kwargs*)

Bases: *shellbot.commands.base.Command*

Closes the space

>>>close = Close(engine=my_engine) >>>shell.load_command(close)

execute(*bot*, *arguments=None*, ***kwargs*)

Closes the space

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

This function should report on progress by sending messages with one or multiple `bot.say("Whatever response")`.

in_direct = False

information_message = u'Close this space'

keyword = u'close'

shellbot.commands.default module

class shellbot.commands.default.**Default**(*engine=None*, ***kwargs*)

Bases: *shellbot.commands.base.Command*

Handles unmatched command

This function looks for a named list and adds participants accordingly. Note that only list with attribute `as_command` set to true are considered.

In other cases, the end user is advised that the command is unknown.

default_message = u'Sorry, I do not know how to handle '{}''

execute(*bot*, *arguments=None*, ***kwargs*)
Handles unmatched command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

Arguments provided should include all of the user input, including the first token that has not been recognised as a valid command.

If arguments match a named list, then items of the list are added as participants to the channel. This applies only: - if the named list has the attribute `as_command` - and if this is not a direct channel (limited to 1:1 interactions)

information_message = u'Handle unmatched commands'

is_hidden = True

keyword = u'*default'

participants_message = u'Adding participants from '{}''

shellbot.commands.echo module

class shellbot.commands.echo.**Echo**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Echoes input string

execute(*bot*, *arguments=None*, ***kwargs*)
Echoes input string

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Echo input string'

is_hidden = True

keyword = u'echo'

shellbot.commands.empty module

class shellbot.commands.empty.**Empty**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Handles empty command

execute(*bot*, *arguments=None*, ***kwargs*)
Handles empty command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Handle empty command'

is_hidden = True

keyword = u'*empty'

shellbot.commands.help module

class shellbot.commands.help.**Help**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Lists available commands and related usage information

allow(*bot*, *command*)
Allows a command for this bot

Parameters

- **bot** (ShellBot) – Can be a direct channel, or a group channel
- **command** (Command) – Can be restricted either to direct or to group channels

Returns True is this command is allowed for this bot, else False

execute(*bot*, *arguments=None*, ***kwargs*)
Lists available commands and related usage information

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Show commands and usage'

keyword = u'help'

usage_message = u'help <command>'

usage_template = u'usage: {}'

shellbot.commands.input module

class shellbot.commands.input.**Input**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Displays input data

execute(*bot*, *arguments=None*, ***kwargs*)
Displays input data

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Display all input'

input_header = u'Input:'

keyword = u'input'

no_input_message = u'There is nothing to display'

shellbot.commands.noop module

class shellbot.commands.noop.**Noop**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Does absolutely nothing

execute(*bot*, *arguments=None*, ***kwargs*)
Does absolutely nothing

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Do absolutely nothing'

is_hidden = True

keyword = u'pass'

shellbot.commands.sleep module

class shellbot.commands.sleep.**Sleep**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Sleeps for a while

DEFAULT_DELAY = 1.0

execute(*bot*, *arguments=None*, ***kwargs*)
Sleeps for a while

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Sleep for a while'

is_hidden = True

keyword = u'sleep'

usage_message = u'sleep <n>'

shellbot.commands.start module

class shellbot.commands.start.**Start**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Restarts the underlying state machine

This command restarts the current state machine. A typical use case is when a person interacts with the bot over a direct channel for the initial gathering of data. For this kind of situation, the person will type `start` each time she initiates a new sequence.

You can check `examples/direct.py` as a practical tutorial.

Note: this command has no effect on a running machine.

Example to load the command in the engine:

```
engine = Engine(commands=['shellbot.commands.start', ...])
```

By default the command is visible only from direct channels. You can change this by configuring an instance before it is given to the engine:

```
start = Start()
start.in_group = True

engine = Engine(commands=[start, ...])
```

Obviously, this command should not be loaded if your use case does not rely on state machines, or if your state machines never end.

execute (*bot*, *arguments=None*, ***kwargs*)

Restarts the underlying state machine

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

This function calls the `restart()` function of the underlying state machine. It also transmits text typed by the end user after the command verb, and any other parameters received from the shell, e.g., attachment, etc.

Note: this command has no effect on a running machine.

in_direct = True

in_group = False

information_message = u'Start a new sequence'

keyword = u'start'

shellbot.commands.step module

class shellbot.commands.step.**Step** (*engine=None*, ***kwargs*)

Bases: *shellbot.commands.base.Command*

Moves underlying state machine to the next step

This command sends an event to the current state machine. This can be handled by the state machine, e.g., a `Steps` instance, for moving forward.

You can check `examples/pushy.py` as a practical tutorial.

Example to load the command in the engine:

```
engine = Engine(commands=['shellbot.commands.step', ...])
```

By default, the command sends the event `next` to the state machine. This can be changed while creating your own command instance, before it is given to the engine. For example:

```
step = Steps(event='super_event')
engine = Engine(commands=[step, ...])
```

Obviously, this command should not be loaded if your use case does not rely on state machines, or if your state machines do not expect events from human beings.

event = 'next'

execute(*bot*, *arguments=None*, ***kwargs*)
Moves underlying state machine to the next step

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

This function calls the `step()` function of the underlying state machine and provides a static event. It also transmits text typed by the end user after the command verb, and any other parameters received from the shell, e.g., attachment, etc.

information_message = u'Move process to next step'

keyword = u'step'

shellbot.commands.update module

class shellbot.commands.update.**Update**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Update input data

execute(*bot*, *arguments=None*, ***kwargs*)
Update input data

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Update input content'

keyword = u'update'

no_arg = u'Thanks to provide the key and the data'

no_input = u'There is nothing to update, input is empty'

ok_msg = u'Update successfuly done'

shellbot.commands.upload module

class shellbot.commands.upload.**Upload**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Handles a bare file upload

execute(*bot*, *attachment*, *url*, *arguments=None*, ***kwargs*)
Handles bare upload

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **attachment** (*str*) – External name of the uploaded file
- **url** (*str*) – The link to fetch actual content
- **arguments** (str or None) – The arguments for this command

feedback_message = u'Thank you for the information shared!'

information_message = u'Handle file upload'

is_hidden = True

keyword = u'*upload'

shellbot.commands.version module

class shellbot.commands.version.**Version**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Displays software version

execute(*bot*, *arguments=None*, ***kwargs*)
Displays software version

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Display software version'

is_hidden = True

keyword = u'version'

Module contents

class shellbot.commands.**Audit**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Checks and changes audit status

In essence, audit starts with the capture of information in real-time, and continues with the replication of information.

A typical use case is the monitoring of interactions happening in a channel, for security reasons, for compliancy or, simply speaking, for traceability.

Audit can be suspended explicitly by channel participants. This allows for some private exchanges that are not audited at all. However, then command is put in the audit log itself, so that people can be queried afterwards on their private interactions. If the parameter off_duration is set, then it is used by a watchdog to restart auditing. Else it is up to channel participants to activate or to de-activate auditing, at will.

The command itself allows for suspending or restarting the audit process. When audit has been activated in a channel, the attribute `audit.switch.<channel_id>` is set to `on` in the context. This can be checked by the observer while handling inbound records.

The audit has to be armed beforehand, and this is checked from the context attribute `audit.has_been_armed`. In normal cases, audit is armed from the underlying space by setting this attribute to True.

already_off_message = u'Chat interactions are already private.'

already_on_message = u'Chat interactions are already audited.'

audit_off(*bot*)
Activates private mode

Parameters **bot** (*Shellbot*) – The bot for this execution

audit_on(*bot*)
Activates audit mode

Parameters **bot** (*Shellbot*) – The bot for this execution

audit_status(*bot*)
Reports on audit status

Parameters **bot** (*Shellbot*) – The bot for this execution

disabled_message = u'Audit has not been enabled.'

execute(*bot*, *arguments=None*, ***kwargs*)
Checks and changes audit status

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (*str*) – either 'on' or 'off'

has_been_enabled
Are we ready for auditing or not?

Return type bool

in_direct = False

information_message = u'Check and change audit status'

keyword = u'audit'

off_duration = 60

off_message = u'Chat interactions are not audited.'

on_bond(*bot*)
Activates audit when a bot joins a channel

on_init()
Registers callback from bot

on_message = u'Chat interactions are currently audited.'

on_off(*bot*)
Triggers watchdog when audit is disabled

temporary_off_message = u'Please note that auditing will restart after {}'

usage_message = u'audit [on|off]'

watchdog(*bot*)
Ensures that audit is restarted

class shellbot.commands.**Command**(*engine=None*, ***kwargs*)
Bases: object

Implements one command

execute(*bot*, *arguments=None*, ***kwargs*)
Executes this command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

The function is invoked with a variable number of arguments. Therefore the need for **kwargs, so that your code is safe in all cases.

The recommended signature for commands that handle textual arguments is the following:

‘‘‘ def execute(self, bot, arguments=None, **kwargs):

... if arguments:

...

‘‘‘

In this situation, arguments contains all text typed after the verb itself. For example, when the command magic is invoked with the string:

```
magic rub the lamp
```

then the related command instance is called like this:

```
magic = shell.command('magic')
magic.execute(bot, arguments='rub the lamp')
```

For commands that can handle file attachments, you could use following approach:

```
def execute(self,
            bot,
            arguments=None,
            attachment=None,
            url=None,
            **kwargs):
    ...
    if url:  # a document has been uploaded with this command
        content = bot.space.download_attachment(url)
        ...
```

Reference information on parameters provided by the shell:

- bot - This is the bot instance for which the command is executed. From this you can update the chat with bot.say(), or access data attached to the bot in bot.store. The engine and all global items can be access with bot.engine.
- arguments - This is a string that contains everything after the command verb. When hello How are you doing? is submitted to the shell, hello is the verb, and How are you doing? are the arguments. This is the regular case. If there is no command hello then the command *default is used instead, and arguments provided are the full line hello How are you doing?.

- `attachment` - When a file has been uploaded, this attribute provides its external name, e.g., `picture024.png`. This can be used in the executed command, if you keep in mind that the same name can be used multiple times in a conversation.
- `url` - When a file has been uploaded, this is the handle by which actual content can be retrieved. Usually, ask the underlying space to get a local copy of the document.

This function should report on progress by sending messages with one or multiple `bot.say("Whatever response")`.

in_direct = True

in_group = True

information_message = None

is_hidden = False

keyword = None

on_init()
Handles extended initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self):
    self.engine.register('stop', self)
```

usage_message = None

class shellbot.commands.**Close**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Closes the space

>>>close = Close(engine=my_engine) >>>shell.load_command(close)

execute(*bot*, *arguments=None*, ***kwargs*)
Closes the space

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

This function should report on progress by sending messages with one or multiple `bot.say("Whatever response")`.

in_direct = False

information_message = u'Close this space'

keyword = u'close'

class shellbot.commands.**Default**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Handles unmatched command

This function looks for a named list and adds participants accordingly. Note that only list with attribute `as_command` set to true are considered.

In other cases, the end user is advised that the command is unknown.

default_message = u'Sorry, I do not know how to handle '{}'"

execute(*bot*, *arguments=None*, ***kwargs*)
Handles unmatched command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

Arguments provided should include all of the user input, including the first token that has not been recognised as a valid command.

If arguments match a named list, then items of the list are added as participants to the channel. This applies only: - if the named list has the attribute as_command - and if this is not a direct channel (limited to 1:1 interactions)

information_message = u'Handle unmatched commands'

is_hidden = True

keyword = u'*default'

participants_message = u"Adding participants from '{}'"

class shellbot.commands.**Echo**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Echoes input string

execute(*bot*, *arguments=None*, ***kwargs*)
Echoes input string

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Echo input string'

is_hidden = True

keyword = u'echo'

class shellbot.commands.**Empty**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Handles empty command

execute(*bot*, *arguments=None*, ***kwargs*)
Handles empty command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Handle empty command'

is_hidden = True

keyword = u'*empty'

class shellbot.commands.**Input**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Displays input data

execute(*bot*, *arguments=None*, ***kwargs*)
Displays input data

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Display all input'

input_header = u'Input:'

keyword = u'input'

no_input_message = u'There is nothing to display'

class shellbot.commands.**Help**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Lists available commands and related usage information

allow(*bot*, *command*)
Allows a command for this bot

Parameters

- **bot** (ShellBot) – Can be a direct channel, or a group channel
- **command** (Command) – Can be restricted either to direct or to group channels

Returns True is this command is allowed for this bot, else False

execute(*bot*, *arguments=None*, ***kwargs*)
Lists available commands and related usage information

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Show commands and usage'

keyword = u'help'

usage_message = u'help <command>'

usage_template = u'usage: {}'

class shellbot.commands.**Noop**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Does absolutely nothing

execute(*bot*, *arguments=None*, ***kwargs*)
Does absolutely nothing

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Do absolutely nothing'

is_hidden = True

keyword = u'pass'

class shellbot.commands.**Sleep**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Sleeps for a while

DEFAULT_DELAY = 1.0

execute(*bot*, *arguments=None*, ***kwargs*)
Sleeps for a while

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Sleep for a while'

is_hidden = True

keyword = u'sleep'

usage_message = u'sleep <n>'

class shellbot.commands.**Step**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Moves underlying state machine to the next step

This command sends an event to the current state machine. This can be handled by the state machine, e.g., a Steps instance, for moving forward.

You can check examples/pushy.py as a practical tutorial.

Example to load the command in the engine:

```
engine = Engine(commands=['shellbot.commands.step', ...])
```

By default, the command sends the event next to the state machine. This can be changed while creating your own command instance, before it is given to the engine. For example:

```
step = Steps(event='super_event')
engine = Engine(commands=[step, ...])
```

Obviously, this command should not be loaded if your use case does not rely on state machines, or if your state machines do not expect events from human beings.

event = 'next'

execute(*bot*, *arguments=None*, ***kwargs*)
Moves underlying state machine to the next step

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

This function calls the step() function of the underlying state machine and provides a static event. It also transmits text typed by the end user after the command verb, and any other parameters received from the shell, e.g., attachment, etc.

information_message = u'Move process to next step'

keyword = u'step'

class shellbot.commands.**Upload**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Handles a bare file upload

execute(*bot*, *attachment*, *url*, *arguments=None*, ***kwargs*)
Handles bare upload

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **attachment** (*str*) – External name of the uploaded file
- **url** (*str*) – The link to fetch actual content
- **arguments** (str or None) – The arguments for this command

feedback_message = u'Thank you for the information shared!'

information_message = u'Handle file upload'

is_hidden = True

keyword = u'*upload'

class shellbot.commands.**Version**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Displays software version

execute(*bot*, *arguments=None*, ***kwargs*)
Displays software version

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Display software version'

is_hidden = True

keyword = u'version'

class shellbot.commands.**Update**(*engine=None*, ***kwargs*)
Bases: *shellbot.commands.base.Command*

Update input data

execute(*bot*, *arguments=None*, ***kwargs*)
Update input data

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

information_message = u'Update input content'

keyword = u'update'

no_arg = u'Thanks to provide the key and the data'

no_input = u'There is nothing to update, input is empty'

ok_msg = u'Update successfuly done'

shellbot.lists package

Submodules

shellbot.lists.base module

class shellbot.lists.base.**List**(*context=None*, ***kwargs*)
Bases: object

Implements an immutable list

This allows easy integration of external reference data such as list of e-mail addresses, etc.

on_init(*items=[]*, ***kwargs*)
Handles extended initialisation parameters

Parameters **items** (*list or set*) – a list of items

Example:

```
list = List(items=['a', 'b', 'c'])
for item in list:
    ...
```

This function should be expanded in sub-class, where necessary.

Module contents

class shellbot.lists.**List**(*context=None*, ***kwargs*)
Bases: object

Implements an immutable list

This allows easy integration of external reference data such as list of e-mail addresses, etc.

on_init(*items=[]*, ***kwargs*)
Handles extended initialisation parameters

Parameters **items** (*list or set*) – a list of items

Example:

```
list = List(items=['a', 'b', 'c'])
for item in list:
    ...
```

This function should be expanded in sub-class, where necessary.

class shellbot.lists.**ListFactory**(*context=None*)
Bases: object

Manages named lists

Example:

```
factory = ListFactory(context=my_context)
factory.configure()
...
my_list = factory.get_list('The Famous Four')
```

apply_to_list(*name*, *apply*)

Handles each item of a named list

Parameters

- **name** (*str*) – designates the list to use
- **apply** (callable) – the function that is applied to each item

This function calls the provided function for each item of a named list.

For example, you could write an alerting system like this:

```
def alert(person):
    number = get_phone_number(person)
    send_sms(important_message, number)

factory.apply_to_list('SupportTeam', alert)
```

Lambda functions are welcome as well. For example, this can be useful for the straightforward addition of participants to a given bot:

```
factory.apply_to_list(name='SupportTeam',
                      apply=lambda x: my_bot.add_participant(x))
```

build_list(*attributes*)

Builds one list

Example in YAML:

```
- name: The Famous Four
  as_command: true
  items:
    - alice@acme.com
    - bob@project.org
    - celine@secret.mil
    - dude@bangkok.travel
```

The as_command parameter is a boolean that indicates if the list can be used as a shell command. When as_command is set to true, the named list appears in the list of shell commands. Members of the list are added to a channel when the name of the list is submitted to the shell.

configure()

Loads lists as defined in context

This function looks for the key lists and below in the context, and creates a dictionary of named lists.

Example configuration in YAML format:

```
lists:

    - name: The Famous Four
      items:
        - alice@acme.com
        - bob@project.org
```

```
        - celine@secret.mil
        - dude@bangkok.travel

    - name: Support Team
      items:
        - service.desk@acme.com
        - supervisor@brother.mil
```

Note that list names are all put to lower case internally, for easy subsequent references. With the previous examples, you can retrieve the first list with *The Famous Four* or with *the famous four*. This is spacially convenient for lists used as commands, when invoked from a mobile device.

get_list(*name*)

Gets a named list

Parameters **name** (*str*) – Name of the target list

Returns an iterator

An empty list is returned when the name is unknown.

Example use case, where an alert is sent to members of a team:

```
for person in factory.get_list('SupportTeam'):
    number = get_phone_number(person)
    send_sms(important_message, number)
```

list_commands()

Lists items that can be invoked as shell commands

Returns an iterator of list names

shellbot.machines package

Submodules

shellbot.machines.base module

class shellbot.machines.base.**Machine**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)

Bases: object

Implements a state machine

The life cycle of a machine can be described as follows:

1.A machine instance is created and configured:

```
a_bot = ShellBot(...)
machine = Machine(bot=a_bot)

machine.set(states=states, transitions=transitions, ...
```

2.The machine is switched on and ticked at regular intervals:

```
machine.start()
```

3.Machine can process more events than ticks:

```
machine.execute('hello world')
```

4.When a machine is expecting data from the chat space, it listens from the `fan` queue used by the shell:

```
engine.fan.put('special command')
```

5.When the machine is coming end of life, resources can be disposed:

```
machine.stop()
```

credit: Alex Bertsch <abertsch@dropbox.com> securitybot/state_machine.py

DEFER_DURATION = 0.0

TICK_DURATION = 0.2

build(*states*, *transitions*, *initial*, *during=None*, *on_enter=None*, *on_exit=None*)

Builds a complete state machine

Parameters

- **states** (*list of str*) – All states supported by this machine
- **transitions** (*list of dict*) – Transitions between states. Each transition is a dictionary. Each dictionary must feature following keys:

  source (str): The source state of the transition target (str): The target state of the transition

  Each dictionary may contain following keys:

  condition (function): A condition that must be true for the transition to occur. If no condition is provided then the state machine will transition on a step.

  action (function): A function to be executed while the transition occurs.

- **initial** (*str*) – The initial state
- **during** (*dict*) – A mapping of states to functions to execute while in that state. Each key should map to a callable function.
- **on_enter** (*dict*) – A mapping of states to functions to execute when entering that state. Each key should map to a callable function.
- **on_exit** (*dict*) – A mapping of states to functions to execute when exiting that state. Each key should map to a callable function.

current_state

Provides current state

Returns State

This function raises AttributeError if it is called before `build()`.

execute(*arguments=None*, ***kwargs*)

Processes data received from the chat

Parameters **arguments** (*str is recommended*) – input to be injected into the state machine

This function can be used to feed the machine asynchronously

get(*key*, *default=None*)

Retrieves the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **default** (*an type that can be serialized*) – default value is the attribute has not been set yet

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

is_running
Determines if this machine is runnning

Returns True or False

on_init(***kwargs*)
Adds to machine initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='my.machine', **kwargs):
    ...
```

on_reset()
Adds processing to machine reset

This function should be expanded in sub-class, where necessary.

Example:

```
def on_reset(self):
    self.sub_machine.reset()
```

on_start()
Adds to machine start

This function is invoked when the machine is started or restarted. It can be expanded in sub-classes where required.

Example:

```
def on_start(self):  # clear bot store on machine start
    self.bot.forget()
```

on_stop()
Adds to machine stop

This function is invoked when the machine is stopped. It can be expanded in sub-classes where required.

Example:

```
def on_stop(self):  # dump bot store on machine stop
    self.bot.publisher.put(
        self.bot.id,
        self.bot.recall('input'))
```

on_tick()
Processes one tick

reset()
Resets a state machine before it is restarted

Returns True if the machine has been actually reset, else False

This function moves a state machine back to its initial state. A typical use case is when you have to recycle a state machine multiple times, like in the following example:

```
if new_cycle():
    machine.reset()
    machine.start()
```

If the machine is running, calling `reset()` will have no effect and you will get False in return. Therefore, if you have to force a reset, you may have to stop the machine first.

Example of forced reset:

```
machine.stop()
machine.reset()
```

restart(***kwargs*)

Restarts the machine

This function is very similar to reset(), except that it also starts the machine on successful reset. Parameters given to it are those that are expected by start().

Note: this function has no effect on a running machine.

run()

Continuously ticks the machine

This function is looping in the background, and calls `step(event='tick')` at regular intervals.

The recommended way for stopping the process is to call the function `stop()`. For example:

```
machine.stop()
```

The loop is also stopped when the parameter `general.switch` is changed in the context. For example:

```
engine.set('general.switch', 'off')
```

set(*key*, *value*)

Remembers the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **value** (*an type that can be serialized*) – new value of the attribute

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

start(*tick=None*, *defer=None*)

Starts the machine

Parameters

- **tick** (*positive number*) – The duration set for each tick (optional)
- **defer** (*positive number*) – wait some seconds before the actual work (optional)

Returns either the process that has been started, or None

This function starts a separate thread to tick the machine in the background.

state(*name*)
Provides a state by name

Parameters **name** (*str*) – The label of the target state

Returns State

This function raises KeyError if an unknown name is provided.

step(***kwargs*)
Brings some life to the state machine

Thanks to `**kwargs`, it is easy to transmit parameters to underlying functions: - `current_state.during(**kwargs)` - `transition.condition(**kwargs)`

Since parameters can vary on complex state machines, you are advised to pay specific attention to the signatures of related functions. If you expect some parameter in a function, use ‘‘kwargs.get()‘‘to get its value safely.

For example, to inject the value of a gauge in the state machine on each tick:

```
def remember(**kwargs):
    gauge = kwargs.get('gauge')
    if gauge:
        db.save(gauge)

during = { 'measuring', remember }

...

machine.build(during=during, ... )

while machine.is_running:
    machine.step(gauge=get_measurement())
```

Or, if you have to transition on a specific threshold for a gauge, you could do:

```
def if_threshold(**kwargs):
    gauge = kwargs.get('gauge')
    if gauge > 20:
        return True
    return False

def raise_alarm():
    mail.post_message()

transitions = [

    {'source': 'normal',
     'target': 'alarm',
     'condition': if_threshold,
     'action': raise_alarm},

     ...

    ]

...

machine.build(transitions=transitions, ... )
```

```
while machine.is_running:
    machine.step(gauge=get_measurement())
```

Shellbot is using this mechanism for itself, and the function can be called at various occasions: - machine tick - This is done at regular intervals in time - input from the chat - Typically, in response to a question - inbound message - Received from subscription, over the network

Following parameters are used for machine ticks: - event='tick' - fixed value

Following parameters are used for chat input: - event='input' - fixed value - arguments - the text that is submitted from the chat

Following parameters are used for subscriptions: - event='inbound' - fixed value - message - the object that has been transmitted

This machine should report on progress by sending messages with one or multiple `self.bot.say("Whatever message")`.

stop()
Stops the machine

This function sends a poison pill to the queue that is read on each tick.

class shellbot.machines.base.**State**(*name*, *during=None*, *on_enter=None*, *on_exit=None*)
Bases: object

Represents a state of the machine

Each state has a function to perform while it's active, when it's entered into, and when it's exited. These functions may be None.

during(***kwargs*)
Does some stuff while in this state

on_enter()
Does some stuf while transitioning into this state

on_exit()
Does some stuff while transitioning out of this state

class shellbot.machines.base.**Transition**(*source*, *target*, *condition=None*, *action=None*)
Bases: object

Represents a transition between two states

Each transition object holds a reference to its source and destination states, as well as the condition function it requires for transitioning and the action to perform upon transitioning.

action()
Does some stuff while transitioning

condition(***kwargs*)
Checks if transition can be triggered

Returns True or False

Condition default to True if none is provided

shellbot.machines.input module

class shellbot.machines.input.**Input**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)

Bases: *shellbot.machines.base.Machine*

Asks for some input

This implements a state machine that can get one piece of input from chat participants. It can ask a question, wait for some input, check provided data and provide guidance when needed.

Example:

```
machine = Input(bot=bot, question="PO Number?", key="order.id")
machine.start()
...
```

In normal operation mode, the machine asks a question in the chat space, then listen for an answer, captures it, and stops.

When no adequate answer is provided, the machine will provide guidance in the chat space after some delay, and ask for a retry. Multiple retries can take place, until correct input is provided, or the machine is timed out.

The machine can also time out after a (possibly) long duration, and send a message in the chat space when giving up.

If correct input is mandatory, no time out will take place and the machine will really need a correct answer to stop.

Data that has been captured can be read from the machine itself. For example:

```
value = machine.get('answer')
```

If the machine is given a key, this is used for feeding the bot store. For example:

```
machine.build(key='my_field', ...)
...

value = bot.recall('input')['my_field']
```

The most straightforward way to process captured data in real-time is to subclass `Input`, like in the following example:

```
class MyInput(Input):

    def on_input(self, value):
        mail.send_message(value)

machine = MyInput(...)
machine.start()
```

ANSWER_MESSAGE = u'Ok, this has been noted'

CANCEL_DELAY = 40.0

CANCEL_MESSAGE = u'Ok, forget about it'

RETRY_DELAY = 20.0

RETRY_MESSAGE = u'Invalid input, please retry'

ask()
Asks the question in the chat space

cancel()
Cancels the question

Used by the state machine on time out

elapsed
Measures time since the question has been asked

Used in the state machine for repeating the question and on time out.

execute(*arguments=None*, ***kwargs*)
Receives data from the chat

Parameters **arguments** (*str*) – data captured from the chat space

This function checks data that is provided, and provides guidance if needed. It can extract information from the provided mask or regular expression, and save it for later use.

filter(*text*)
Filters data from user input

Parameters **text** (*str*) – Text coming from the chat space

Returns Data to be captured, or None

If a mask is provided, or a regular expression, they are used to extract useful information from provided data.

Example to read a PO mumber:

```
machine.build(mask='9999A', ...)
...

po = machine.filter('PO Number is 2413v')
assert po == '2413v'
```

listen()
Listens for data received from the chat space

This function starts a separate process to scan the `bot.fan` queue until time out.

on_inbound(***kwargs*)
Updates the chat on inbound message

on_init(*question=None*, *question_content=None*, *mask=None*, *regex=None*, *on_answer=None*, *on_answer_content=None*, *on_answer_file=None*, *on_retry=None*, *on_retry_content=None*, *on_retry_file=None*, *retry_delay=None*, *on_cancel=None*, *on_cancel_content=None*, *on_cancel_file=None*, *cancel_delay=None*, *is_mandatory=False*, *key=None*, ***kwargs*)
Asks for some input

Parameters

- **question** (*str*) – Message to ask for some input
- **question_content** (*str*) – Rich message to ask for some input
- **mask** (*str*) – A mask to filter the input
- **regex** (*str*) – A regular expression to filter the input
- **on_answer** (*str*) – Message on successful data capture

- **on_answer_content** (*str in Markdown or HTML format*) – Rich message on successful data capture
- **on_answer_file** (*str*) – File to be uploaded on successful data capture
- **on_retry** (*str*) – Message to provide guidance and ask for retry
- **on_retry_content** (*str in Markdown or HTML format*) – Rich message on retry
- **on_retry_file** (*str*) – File to be uploaded on retry
- **retry_delay** (*int*) – Repeat the on_retry message after this delay in seconds
- **on_cancel** (*str*) – Message on time out
- **on_cancel_content** (*str in Markdown or HTML format*) – Rich message on time out
- **on_cancel_file** (*str*) – File to be uploaded on time out
- **is_mandatory** (*boolean*) – If the bot will insist and never give up
- **cancel_delay** (*int*) – Give up on this input after this delay in seconds
- **key** (*str*) – The label associated with data captured in bot store

If a mask is provided, it is used to filter provided input. Use following conventions to build the mask:

- `A` - Any kind of unicode symbol such as `g` or `ç`
- `9` - A digit such as `0` or `2`
- **+ - When following # or 9, indicates optional extensions** of the same type
- **Any other symbol, including punctuation or white space, has to match** exactly.

For example:

- `9999A` will match 4 digits and 1 additional character
- `#9-A+` will match `#3-June 2017`

Alternatively, you can provide a regular expression (regex) to extract useful information from the input.

You can use almost every regular expression that is supported by python. If parenthesis are used, the function returns the first matching group.

For example, you can capture an identifier with a given prefix:

```
machine.build(question="What is the identifier?",
              regex=r'ID-\d\w\d+', ...)
...

id = machine.filter('The id is ID-1W27 I believe')
assert id == 'ID-1W27'
```

As a grouping example, you can capture a domain name by asking for some e-mail address like this:

```
machine.build(question="please enter your e-mail address",
              regex=r'@([\w.]+)', ...)
...

domain_name = machine.filter('my address is foo.bar@acme.com')
assert domain_name == 'acme.com'
```

on_input(*value*, ***kwargs*)
Processes input data

Parameters **value** (*str*) – data that has been captured

This function is called as soon as some input has been captured. It can be overlaid in subclass, as in the following example:

```
class MyInput(Input):

    def on_input(self, value):
        mail.send_message(value)

machine = MyInput(...)
machine.start()
```

The extra parameters wil be used in case of attachment with the value.

receive()
Receives data from the chat space

This function implements a loop until some data has been actually captured, or until the state machine stops for some reason.

The loop is also stopped when the parameter `general.switch` is changed in the context. For example:

```
engine.set('general.switch', 'off')
```

say_answer(*input*)
Responds on correct capture

Parameters **input** (*str*) – the text that has been noted

say_cancel()
Says that input has been timed out

say_retry()
Provides guidance on retry

search_expression(*regex*, *text*)
Searches for a regular expression in text

Parameters

- **regex** (*str*) – A regular expression to be matched
- **text** (*str*) – The string from the chat space

Returns either the matching expression, or None

You can use almost every regular expression that is supported by python. If parenthesis are used, the function returns the first matching group.

For example, you can capture an identifier with a given prefix:

```
machine.build(question="What is the identifier?",
              regex=r'ID-\d\w\d+', ...)
...

id = machine.filter('The id is ID-1W27 I believe')
assert id == 'ID-1W27'
```

As a grouping example, you can capture a domain name by asking for some e-mail address like this:

```
machine.build(question="please enter your e-mail address",
              regex=r'@([\w.]+)', ...)
...

domain_name = machine.filter('my address is foo.bar@acme.com')
assert domain_name == 'acme.com'
```

search_mask(*mask*, *text*)

Searches for structured data in text

Parameters

- **mask** (*str*) – A simple expression to be searched
- **text** (*str*) – The string from the chat space

Returns either the matching expression, or None

Use following conventions to build the mask:

- `A` - Any kind of unicode symbol, such as `g` or `ç`
- `9` - A digit, such as `0` or `2`
- **+ - When following # or 9, indicates optional extensions** of the same type
- **Any other symbol, including punctuation or white space, has to match** exactly.

Some mask examples:

- `9999A` will match 4 digits and 1 additional character
- `#9-A+` will match `#3-June 2017`

Example to read a PO mumber:

```
machine.build(question="What is the PO number?",
              mask='9999A', ...)
...

po = machine.filter('PO Number is 2413v')
assert po == '2413v'
```

shellbot.machines.menu module

class shellbot.machines.menu.**Menu**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)

Bases: *shellbot.machines.input.Input*

Selects among multiple options

This implements a state machine that can capture a choice from chat participants. It can ask a question, wait for some input, check provided data and provide guidance when needed.

Example:

```
machine = Menu(bot=bot,
               question="What would you prefer?",
               options=["Some starter and then main course",
                        "Main course and sweety dessert"])
machine.start()
...
```

```
if machine.get('answer') == 1:
    prepare_appetizer()
    prepare_main_course()

if machine.get('answer') == 2:
    prepare_main_course()
    prepare_some_cake()
```

In normal operation mode, the machine asks a question in the chat space, then listen for an answer, captures it, and stops.

When no adequate answer is provided, the machine will provide guidance in the chat space after some delay, and ask for a retry. Multiple retries can take place, until correct input is provided, or the machine is timed out.

The machine can also time out after a (possibly) long duration, and send a message in the chat space when giving up.

If correct input is mandatory, no time out will take place and the machine will really need a correct answer to stop.

Data that has been captured can be read from the machine itself. For example:

```
value = machine.get('answer')
```

If the machine is given a key, this is used for feeding the bot store. For example:

```
machine.build(key='my_field', ...)
...

value = bot.recall('input')['my_field']
```

The most straightforward way to process captured data in real-time is to subclass `Menu`, like in the following example:

```
class MyMenu(Menu):

    def on_input(self, value):
        do_something_with(value)

machine = MyMenu(...)
machine.start()
```

RETRY_MESSAGE = u'Invalid input, please enter your choice as a number'

ask()

Asks which menu option to select

If a bare question is provided, then text is added to list all available options.

If a rich question is provided, then we assume that it also contains a representation of menu options and displays it 'as-is'.

filter(*text*)

Filters data from user input

Parameters **text** (*str*) – Text coming from the chat space

Returns Text of the selected option, or None

on_init (*options=[]*, ***kwargs*)
Selects among multiple options

Parameters

- **question** (*str*) – Message to ask for some input (mandatory)
- **options** (*list of str*) – The options of the menu
- **on_answer** (*str*) – Message on successful data capture
- **on_answer_content** (*str in Markdown or HTML format*) – Rich message on successful data capture
- **on_answer_file** (*str*) – File to be uploaded on successful data capture
- **on_retry** (*str*) – Message to provide guidance and ask for retry
- **on_retry_content** (*str in Markdown or HTML format*) – Rich message on retry
- **on_retry_file** (*str*) – File to be uploaded on retry
- **retry_delay** (*int*) – Repeat the on_retry message after this delay in seconds
- **on_cancel** (*str*) – Message on time out
- **on_cancel_content** (*str in Markdown or HTML format*) – Rich message on time out
- **on_cancel_file** (*str*) – File to be uploaded on time out
- **is_mandatory** (*boolean*) – If the bot will insist and never give up
- **cancel_delay** (*int*) – Give up on this input after this delay in seconds
- **key** (*str*) – The label associated with data captured in bot store

shellbot.machines.sequence module

class shellbot.machines.sequence.**Sequence** (*bot=None*, *machines=None*, ***kwargs*)
Bases: object

Implements a sequence of multiple machines

This implements one state machine that is actually a combination of multiple sub-machines, ran in sequence. When one sub-machine stops, the next one is activated.

Example:

```
input_1 = Input( ... )
input_2 = Input( ... )
sequence = Sequence([input_1, input_2])
sequence.start()
```

In this example, the first machine is started, then when it ends the second machine is triggered.

get (*key*, *default=None*)
Retrieves the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance

- **default** (*an type that can be serialized*) – default value is the attribute has not been set yet

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

is_running
Determines if this machine is runnning

Returns True or False

on_init(***kwargs*)
Adds to machine initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='my.machine', **kwargs):
    ...
```

on_reset()
Adds processing to machine reset

This function should be expanded in sub-class, where necessary.

reset()
Resets a state machine before it is restarted

Returns True if the machine has been actually reset, else False

This function moves a state machine back to its initial state. A typical use case is when you have to recycle a state machine multiple times, like in the following example:

```
if new_cycle():
    machine.reset()
    machine.start()
```

If the machine is running, calling `reset()` will have no effect and you will get False in return. Therefore, if you have to force a reset, you may have to stop the machine first.

Example of forced reset:

```
machine.stop()
machine.reset()
```

run()
Continuously ticks the sequence

This function is looping in the background, and calls the function `step()` at regular intervals.

The loop is stopped when the parameter `general.switch` is changed in the context. For example:

```
bot.context.set('general.switch', 'off')
```

set(*key*, *value*)
Remembers the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **value** (*an type that can be serialized*) – new value of the attribute

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

start()
Starts the sequence

Returns either the process that has been started, or None

This function starts a separate thread to run machines in the background.

stop()
Stops the sequence

This function stops the underlying machine and breaks the sequence.

shellbot.machines.steps module

class shellbot.machines.steps.**Step**(*attributes*, *index*)
Bases: object

Represents a step in a linear process

say(*bot*)
Reports on this step

Parameters **bot** (ShellBot) – the bit to use

This function posts to the chat space some information on this step.

Example:

```
step.say(bot)
```

stop()
Stops a step

trigger(*bot*)
Triggers a step

Parameters **bot** (ShellBot) – the bot to use

This function does everything that is coming with a step: - send a message to the chat space, - maybe in MarkDown or HTML, - maybe with some attachment, - add participants to the channel, - reset and start a state machine

Example:

```
step.trigger(bot)
```

class shellbot.machines.steps.**Steps**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)
Bases: *shellbot.machines.base.Machine*

Implements a linear process with multiple steps

This implements a state machine that appears as a phased process to chat participants. On each, it can add new participants, display some information, and run a child state machine..

For example, to run an escalation process:

```
po_input = Input( ... )
details_input = Input( ... )

decision_menu = Menu( ...)

steps = [

    {
        'label': u'Level 1',
        'message': u'Initial capture of information',
        'machine': Sequence([po_input, details_input]),
    },

    {
        'label': u'Level 2',
        'message': u'Escalation to technical experts',
    },

    {
        'label': u'Level 3',
        'message': u'Escalation to decision stakeholders',
        'participants': 'bob@acme.com',
        'machine': decision_menu,
    },

    {
        'label': u'Terminated',
        'message': u'Process is closed, yet conversation can continue',
    },

]
machine = Steps(bot=bot, steps=steps)
machine.start()
...
```

current_step
Gets current step

Returns current step, or None

if_end(***kwargs*)
Checks if all steps have been used

Since this function is an integral part of the state machine, it should be triggered via a call of the `step()` member function.

For example:

```
machine.step(event='tick')
```

if_next(***kwargs*)
Checks if next step should be engaged

Parameters **event** (*str*) – a label of event submitted to the machine

This function is used by the state machine for testing the transition to the next available step in the process.

Since this function is an integral part of the state machine, it should be triggered via a call of the `step()` member function.

For example:

```
machine.step(event='next')
```

if_ready(***kwargs*)
Checks if state machine can engage on first step

To be overlaid in sub-class where complex initialization activities are required.

Example:

```
class MyMachine(Machine):

    def if_ready(self, **kwargs):
        if kwargs.get('phase') == 'warming up':
            return False
        else:
            return True
```

next_step()
Moves to next step

Returns current step, or None

This function loads and runs the next step in the process, if any. If all steps have been consumed it returns None.

If a sub-machine is running at current step, it is stopped before moving to the next step.

on_init(*steps=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **steps** (*list of Step or list of dict*) – The steps for this process

on_reset()
Restore initial state of this machine

If a sub-machine is running at current step, it is stopped first.

step_has_completed(***kwargs*)
Checks if the machine for this step has finished its job

Returns False if the machine is still running, True otherwise

Module contents

class shellbot.machines.**Input**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)
Bases: *shellbot.machines.base.Machine*

Asks for some input

This implements a state machine that can get one piece of input from chat participants. It can ask a question, wait for some input, check provided data and provide guidance when needed.

Example:

```
machine = Input(bot=bot, question="PO Number?", key="order.id")
machine.start()
...
```

In normal operation mode, the machine asks a question in the chat space, then listen for an answer, captures it, and stops.

When no adequate answer is provided, the machine will provide guidance in the chat space after some delay, and ask for a retry. Multiple retries can take place, until correct input is provided, or the machine is timed out.

The machine can also time out after a (possibly) long duration, and send a message in the chat space when giving up.

If correct input is mandatory, no time out will take place and the machine will really need a correct answer to stop.

Data that has been captured can be read from the machine itself. For example:

```
value = machine.get('answer')
```

If the machine is given a key, this is used for feeding the bot store. For example:

```
machine.build(key='my_field', ...)
...

value = bot.recall('input')['my_field']
```

The most straightforward way to process captured data in real-time is to subclass `Input`, like in the following example:

```
class MyInput(Input):

    def on_input(self, value):
        mail.send_message(value)

machine = MyInput(...)
machine.start()
```

ANSWER_MESSAGE = u'Ok, this has been noted'

CANCEL_DELAY = 40.0

CANCEL_MESSAGE = u'Ok, forget about it'

RETRY_DELAY = 20.0

RETRY_MESSAGE = u'Invalid input, please retry'

ask()
Asks the question in the chat space

cancel()
Cancels the question

Used by the state machine on time out

elapsed
Measures time since the question has been asked

Used in the state machine for repeating the question and on time out.

execute(*arguments=None*, ***kwargs*)
Receives data from the chat

Parameters **arguments** (*str*) – data captured from the chat space

This function checks data that is provided, and provides guidance if needed. It can extract information from the provided mask or regular expression, and save it for later use.

filter(*text*)
Filters data from user input

Parameters **text** (*str*) – Text coming from the chat space

Returns Data to be captured, or None

If a mask is provided, or a regular expression, they are used to extract useful information from provided data.

Example to read a PO mumber:

```
machine.build(mask='9999A', ...)
...

po = machine.filter('PO Number is 2413v')
assert po == '2413v'
```

listen()
Listens for data received from the chat space

This function starts a separate process to scan the `bot.fan` queue until time out.

on_inbound(***kwargs*)
Updates the chat on inbound message

on_init(*question=None*, *question_content=None*, *mask=None*, *regex=None*, *on_answer=None*, *on_answer_content=None*, *on_answer_file=None*, *on_retry=None*, *on_retry_content=None*, *on_retry_file=None*, *retry_delay=None*, *on_cancel=None*, *on_cancel_content=None*, *on_cancel_file=None*, *cancel_delay=None*, *is_mandatory=False*, *key=None*, ***kwargs*)
Asks for some input

Parameters

- **question** (*str*) – Message to ask for some input
- **question_content** (*str*) – Rich message to ask for some input
- **mask** (*str*) – A mask to filter the input
- **regex** (*str*) – A regular expression to filter the input
- **on_answer** (*str*) – Message on successful data capture
- **on_answer_content** (*str in Markdown or HTML format*) – Rich message on successful data capture
- **on_answer_file** (*str*) – File to be uploaded on successful data capture
- **on_retry** (*str*) – Message to provide guidance and ask for retry
- **on_retry_content** (*str in Markdown or HTML format*) – Rich message on retry
- **on_retry_file** (*str*) – File to be uploaded on retry
- **retry_delay** (*int*) – Repeat the on_retry message after this delay in seconds
- **on_cancel** (*str*) – Message on time out
- **on_cancel_content** (*str in Markdown or HTML format*) – Rich message on time out
- **on_cancel_file** (*str*) – File to be uploaded on time out
- **is_mandatory** (*boolean*) – If the bot will insist and never give up

- **cancel_delay** (*int*) – Give up on this input after this delay in seconds
- **key** (*str*) – The label associated with data captured in bot store

If a mask is provided, it is used to filter provided input. Use following conventions to build the mask:

- `A` - Any kind of unicode symbol such as `g` or `ç`
- `9` - A digit such as `0` or `2`
- **`+` - When following `#` or `9`, indicates optional extensions** of the same type
- **Any other symbol, including punctuation or white space, has to match** exactly.

For example:

- `9999A` will match 4 digits and 1 additional character
- `#9-A+` will match `#3-June 2017`

Alternatively, you can provide a regular expression (regex) to extract useful information from the input.

You can use almost every regular expression that is supported by python. If parenthesis are used, the function returns the first matching group.

For example, you can capture an identifier with a given prefix:

```
machine.build(question="What is the identifier?",
              regex=r'ID-\d\w\d+', ...)
...

id = machine.filter('The id is ID-1W27 I believe')
assert id == 'ID-1W27'
```

As a grouping example, you can capture a domain name by asking for some e-mail address like this:

```
machine.build(question="please enter your e-mail address",
              regex=r'@([\w.]+)', ...)
...

domain_name = machine.filter('my address is foo.bar@acme.com')
assert domain_name == 'acme.com'
```

on_input(*value*, ***kwargs*)

Processes input data

Parameters **value** (*str*) – data that has been captured

This function is called as soon as some input has been captured. It can be overlaid in subclass, as in the following example:

```
class MyInput(Input):

    def on_input(self, value):
        mail.send_message(value)

machine = MyInput(...)
machine.start()
```

The extra parameters wil be used in case of attachment with the value.

receive()

Receives data from the chat space

This function implements a loop until some data has been actually captured, or until the state machine stops for some reason.

The loop is also stopped when the parameter `general.switch` is changed in the context. For example:

```
engine.set('general.switch', 'off')
```

say_answer(*input*)
Responds on correct capture

Parameters **input** (*str*) – the text that has been noted

say_cancel()
Says that input has been timed out

say_retry()
Provides guidance on retry

search_expression(*regex*, *text*)
Searches for a regular expression in text

Parameters

- **regex** (*str*) – A regular expression to be matched
- **text** (*str*) – The string from the chat space

Returns either the matching expression, or None

You can use almost every regular expression that is supported by python. If parenthesis are used, the function returns the first matching group.

For example, you can capture an identifier with a given prefix:

```
machine.build(question="What is the identifier?",
              regex=r'ID-\d\w\d+', ...)
...

id = machine.filter('The id is ID-1W27 I believe')
assert id == 'ID-1W27'
```

As a grouping example, you can capture a domain name by asking for some e-mail address like this:

```
machine.build(question="please enter your e-mail address",
              regex=r'@([\w.]+)', ...)
...

domain_name = machine.filter('my address is foo.bar@acme.com')
assert domain_name == 'acme.com'
```

search_mask(*mask*, *text*)
Searches for structured data in text

Parameters

- **mask** (*str*) – A simple expression to be searched
- **text** (*str*) – The string from the chat space

Returns either the matching expression, or None

Use following conventions to build the mask:

•`A` - Any kind of unicode symbol, such as `g` or `ç`

- `9` - A digit, such as `0` or `2`
- **`+` - When following `#` or `9`, indicates optional extensions** of the same type
- **Any other symbol, including punctuation or white space, has to match** exactly.

Some mask examples:

- `9999A` will match 4 digits and 1 additional character
- `#9-A+` will match `#3-June 2017`

Example to read a PO mumber:

```
machine.build(question="What is the PO number?",
              mask='9999A', ...)
...

po = machine.filter('PO Number is 2413v')
assert po == '2413v'
```

class `shellbot.machines.`**`Machine`**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)

Bases: `object`

Implements a state machine

The life cycle of a machine can be described as follows:

1.A machine instance is created and configured:

```
a_bot = ShellBot(...)
machine = Machine(bot=a_bot)

machine.set(states=states, transitions=transitions, ...
```

2.The machine is switched on and ticked at regular intervals:

```
machine.start()
```

3.Machine can process more events than ticks:

```
machine.execute('hello world')
```

4.When a machine is expecting data from the chat space, it listens from the `fan` queue used by the shell:

```
engine.fan.put('special command')
```

5.When the machine is coming end of life, resources can be disposed:

```
machine.stop()
```

credit: Alex Bertsch <abertsch@dropbox.com> securitybot/state_machine.py

`DEFER_DURATION = 0.0`

`TICK_DURATION = 0.2`

`build`(*states*, *transitions*, *initial*, *during=None*, *on_enter=None*, *on_exit=None*)

Builds a complete state machine

Parameters

- **`states`** (*`list of str`*) – All states supported by this machine

- **transitions** (*list of dict*) – Transitions between states. Each transition is a dictionary. Each dictionary must feature following keys:

  source (str): The source state of the transition target (str): The target state of the transition

  Each dictionary may contain following keys:

  condition (function): A condition that must be true for the transition to occur. If no condition is provided then the state machine will transition on a step.

  action (function): A function to be executed while the transition occurs.

- **initial** (*str*) – The initial state
- **during** (*dict*) – A mapping of states to functions to execute while in that state. Each key should map to a callable function.
- **on_enter** (*dict*) – A mapping of states to functions to execute when entering that state. Each key should map to a callable function.
- **on_exit** (*dict*) – A mapping of states to functions to execute when exiting that state. Each key should map to a callable function.

current_state
Provides current state

Returns State

This function raises AttributeError if it is called before `build()`.

execute(*arguments=None*, ***kwargs*)
Processes data received from the chat

Parameters **arguments** (*str is recommended*) – input to be injected into the state machine

This function can be used to feed the machine asynchronously

get(*key*, *default=None*)
Retrieves the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **default** (*an type that can be serialized*) – default value is the attribute has not been set yet

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

is_running
Determines if this machine is runnning

Returns True or False

on_init(***kwargs*)
Adds to machine initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='my.machine', **kwargs):
    ...
```

on_reset()
Adds processing to machine reset

This function should be expanded in sub-class, where necessary.

Example:

```
def on_reset(self):
    self.sub_machine.reset()
```

on_start()
Adds to machine start

This function is invoked when the machine is started or restarted. It can be expanded in sub-classes where required.

Example:

```
def on_start(self):  # clear bot store on machine start
    self.bot.forget()
```

on_stop()
Adds to machine stop

This function is invoked when the machine is stopped. It can be expanded in sub-classes where required.

Example:

```
def on_stop(self):  # dump bot store on machine stop
    self.bot.publisher.put(
        self.bot.id,
        self.bot.recall('input'))
```

on_tick()
Processes one tick

reset()
Resets a state machine before it is restarted

Returns True if the machine has been actually reset, else False

This function moves a state machine back to its initial state. A typical use case is when you have to recycle a state machine multiple times, like in the following example:

```
if new_cycle():
    machine.reset()
    machine.start()
```

If the machine is running, calling `reset()` will have no effect and you will get False in return. Therefore, if you have to force a reset, you may have to stop the machine first.

Example of forced reset:

```
machine.stop()
machine.reset()
```

restart(***kwargs*)
Restarts the machine

This function is very similar to reset(), except that it also starts the machine on successful reset. Parameters given to it are those that are expected by start().

Note: this function has no effect on a running machine.

run()

Continuously ticks the machine

This function is looping in the background, and calls `step(event='tick')` at regular intervals.

The recommended way for stopping the process is to call the function `stop()`. For example:

```
machine.stop()
```

The loop is also stopped when the parameter `general.switch` is changed in the context. For example:

```
engine.set('general.switch', 'off')
```

set(*key*, *value*)

Remembers the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **value** (*an type that can be serialized*) – new value of the attribute

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

start(*tick=None*, *defer=None*)

Starts the machine

Parameters

- **tick** (*positive number*) – The duration set for each tick (optional)
- **defer** (*positive number*) – wait some seconds before the actual work (optional)

Returns either the process that has been started, or None

This function starts a separate thread to tick the machine in the background.

state(*name*)

Provides a state by name

Parameters **name** (*str*) – The label of the target state

Returns State

This function raises KeyError if an unknown name is provided.

step(***kwargs*)

Brings some life to the state machine

Thanks to `**kwargs`, it is easy to transmit parameters to underlying functions: - `current_state.during(**kwargs)` - `transition.condition(**kwargs)`

Since parameters can vary on complex state machines, you are advised to pay specific attention to the signatures of related functions. If you expect some parameter in a function, use ‘‘kwargs.get()‘‘to get its value safely.

For example, to inject the value of a gauge in the state machine on each tick:

```
def remember(**kwargs):
    gauge = kwargs.get('gauge')
    if gauge:
        db.save(gauge)

during = { 'measuring', remember }

...

machine.build(during=during, ... )

while machine.is_running:
    machine.step(gauge=get_measurement())
```

Or, if you have to transition on a specific threshold for a gauge, you could do:

```
def if_threshold(**kwargs):
    gauge = kwargs.get('gauge')
    if gauge > 20:
        return True
    return False

def raise_alarm():
    mail.post_message()

transitions = [

    {'source': 'normal',
     'target': 'alarm',
     'condition': if_threshold,
     'action': raise_alarm},

     ...

    ]

...

machine.build(transitions=transitions, ... )

while machine.is_running:
    machine.step(gauge=get_measurement())
```

Shellbot is using this mechanism for itself, and the function can be called at various occasions: - machine tick - This is done at regular intervals in time - input from the chat - Typically, in response to a question - inbound message - Received from subscription, over the network

Following parameters are used for machine ticks: - event='tick' - fixed value

Following parameters are used for chat input: - event='input' - fixed value - arguments - the text that is submitted from the chat

Following parameters are used for subscriptions: - event='inbound' - fixed value - message - the object that has been transmitted

This machine should report on progress by sending messages with one or multiple `self.bot.say("Whatever message")`.

stop()

Stops the machine

This function sends a poison pill to the queue that is read on each tick.

class shellbot.machines.**Sequence**(*bot=None*, *machines=None*, ***kwargs*)
Bases: object

Implements a sequence of multiple machines

This implements one state machine that is actually a combination of multiple sub-machines, ran in sequence. When one sub-machine stops, the next one is activated.

Example:

```
input_1 = Input( ... )
input_2 = Input( ... )
sequence = Sequence([input_1, input_2])
sequence.start()
```

In this example, the first machine is started, then when it ends the second machine is triggered.

get(*key*, *default=None*)
Retrieves the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **default** (*an type that can be serialized*) – default value is the attribute has not been set yet

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

is_running
Determines if this machine is runnning

Returns True or False

on_init(***kwargs*)
Adds to machine initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='my.machine', **kwargs):
    ...
```

on_reset()
Adds processing to machine reset

This function should be expanded in sub-class, where necessary.

reset()
Resets a state machine before it is restarted

Returns True if the machine has been actually reset, else False

This function moves a state machine back to its initial state. A typical use case is when you have to recycle a state machine multiple times, like in the following example:

```
if new_cycle():
    machine.reset()
    machine.start()
```

If the machine is running, calling `reset()` will have no effect and you will get False in return. Therefore, if you have to force a reset, you may have to stop the machine first.

Example of forced reset:

```
machine.stop()
machine.reset()
```

run()
Continuously ticks the sequence

This function is looping in the background, and calls the function `step()` at regular intervals.

The loop is stopped when the parameter `general.switch` is changed in the context. For example:

```
bot.context.set('general.switch', 'off')
```

set(*key*, *value*)
Remembers the value of one key

Parameters

- **key** (*str*) – one attribute of this state machine instance
- **value** (*an type that can be serialized*) – new value of the attribute

This function can be used across multiple processes, so that a consistent view of the state machine is provided.

start()
Starts the sequence

Returns either the process that has been started, or None

This function starts a separate thread to run machines in the background.

stop()
Stops the sequence

This function stops the underlying machine and breaks the sequence.

class shellbot.machines.**Steps**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)
Bases: *shellbot.machines.base.Machine*

Implements a linear process with multiple steps

This implements a state machine that appears as a phased process to chat participants. On each, it can add new participants, display some information, and run a child state machine..

For example, to run an escalation process:

```
po_input = Input( ... )
details_input = Input( ... )

decision_menu = Menu( ...)

steps = [

    {
        'label': u'Level 1',
        'message': u'Initial capture of information',
```

```
        'machine': Sequence([po_input, details_input]),
    },

    {
        'label': u'Level 2',
        'message': u'Escalation to technical experts',
    },

    {
        'label': u'Level 3',
        'message': u'Escalation to decision stakeholders',
        'participants': 'bob@acme.com',
        'machine': decision_menu,
    },

    {
        'label': u'Terminated',
        'message': u'Process is closed, yet conversation can continue',
    },

]
machine = Steps(bot=bot, steps=steps)
machine.start()
...
```

current_step
Gets current step

Returns current step, or None

if_end(***kwargs*)
Checks if all steps have been used

Since this function is an integral part of the state machine, it should be triggered via a call of the `step()` member function.

For example:

```
machine.step(event='tick')
```

if_next(***kwargs*)
Checks if next step should be engaged

Parameters **event** (*str*) – a label of event submitted to the machine

This function is used by the state machine for testing the transition to the next available step in the process.

Since this function is an integral part of the state machine, it should be triggered via a call of the `step()` member function.

For example:

```
machine.step(event='next')
```

if_ready(***kwargs*)
Checks if state machine can engage on first step

To be overlaid in sub-class where complex initialization activities are required.

Example:

```
class MyMachine(Machine):

    def if_ready(self, **kwargs):
        if kwargs.get('phase') == 'warming up':
            return False
        else:
            return True
```

next_step()
Moves to next step

Returns current step, or None

This function loads and runs the next step in the process, if any. If all steps have been consumed it returns None.

If a sub-machine is running at current step, it is stopped before moving to the next step.

on_init(*steps=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **steps** (*list of Step or list of dict*) – The steps for this process

on_reset()
Restore initial state of this machine

If a sub-machine is running at current step, it is stopped first.

step_has_completed(***kwargs*)
Checks if the machine for this step has finished its job

Returns False if the machine is still running, True otherwise

class shellbot.machines.**Menu**(*bot=None*, *states=None*, *transitions=None*, *initial=None*, *during=None*, *on_enter=None*, *on_exit=None*, ***kwargs*)
Bases: *shellbot.machines.input.Input*

Selects among multiple options

This implements a state machine that can capture a choice from chat participants. It can ask a question, wait for some input, check provided data and provide guidance when needed.

Example:

```
machine = Menu(bot=bot,
               question="What would you prefer?",
               options=["Some starter and then main course",
                        "Main course and sweety dessert"])
machine.start()
...

if machine.get('answer') == 1:
    prepare_appetizer()
    prepare_main_course()

if machine.get('answer') == 2:
    prepare_main_course()
    prepare_some_cake()
```

In normal operation mode, the machine asks a question in the chat space, then listen for an answer, captures it, and stops.

When no adequate answer is provided, the machine will provide guidance in the chat space after some delay, and ask for a retry. Multiple retries can take place, until correct input is provided, or the machine is timed out.

The machine can also time out after a (possibly) long duration, and send a message in the chat space when giving up.

If correct input is mandatory, no time out will take place and the machine will really need a correct answer to stop.

Data that has been captured can be read from the machine itself. For example:

```
value = machine.get('answer')
```

If the machine is given a key, this is used for feeding the bot store. For example:

```
machine.build(key='my_field', ...)
...

value = bot.recall('input')['my_field']
```

The most straightforward way to process captured data in real-time is to subclass `Menu`, like in the following example:

```
class MyMenu(Menu):

    def on_input(self, value):
        do_something_with(value)

machine = MyMenu(...)
machine.start()
```

RETRY_MESSAGE = u'Invalid input, please enter your choice as a number'

ask()
Asks which menu option to select

If a bare question is provided, then text is added to list all available options.

If a rich question is provided, then we assume that it also contains a representation of menu options and displays it 'as-is'.

filter(*text*)
Filters data from user input

Parameters **text** (*str*) – Text coming from the chat space

Returns Text of the selected option, or None

on_init(*options=[]*, ***kwargs*)
Selects among multiple options

Parameters

- **question** (*str*) – Message to ask for some input (mandatory)
- **options** (*list of str*) – The options of the menu
- **on_answer** (*str*) – Message on successful data capture
- **on_answer_content** (*str in Markdown or HTML format*) – Rich message on successful data capture
- **on_answer_file** (*str*) – File to be uploaded on successful data capture

- **on_retry** (*str*) – Message to provide guidance and ask for retry
- **on_retry_content** (*str in Markdown or HTML format*) – Rich message on retry
- **on_retry_file** (*str*) – File to be uploaded on retry
- **retry_delay** (*int*) – Repeat the on_retry message after this delay in seconds
- **on_cancel** (*str*) – Message on time out
- **on_cancel_content** (*str in Markdown or HTML format*) – Rich message on time out
- **on_cancel_file** (*str*) – File to be uploaded on time out
- **is_mandatory** (*boolean*) – If the bot will insist and never give up
- **cancel_delay** (*int*) – Give up on this input after this delay in seconds
- **key** (*str*) – The label associated with data captured in bot store

shellbot.routes package

Submodules

shellbot.routes.base module

class shellbot.routes.base.**Route**(*context=None*, ***kwargs*)
Bases: object

Implements one route

delete()

get(***kwargs*)

post()

put()

route = None

shellbot.routes.notifier module

class shellbot.routes.notifier.**NoQueue**
Bases: object

put(*item=None*)

class shellbot.routes.notifier.**Notifier**(*context=None*, ***kwargs*)
Bases: *shellbot.routes.base.Route*

Notifies a queue on web request

>>>queue = Queue() >>>route = Notifier(route='/notify', queue=queue, notification='hello')

When the route is requested over the web, the notification is pushed to the queue.

>>>queue.get() 'hello'

Notification is triggered on GET, POST, PUT and DELETE verbs.

delete()

get(***kwargs*)

notification = None

notify()

post()

put()

queue = <shellbot.routes.notifier.NoQueue object>

route = '/notify'

shellbot.routes.text module

class shellbot.routes.text.**Text**(*context=None*, ***kwargs*)

Bases: *shellbot.routes.base.Route*

Implements a static web page

>>>page = "<html> ... </html>" >>>route = text(route='/index', page=page)

When the route is requested over the web, static content is provided in return.

>>>route.get() "<html> ... </html>"

This class handles only GET requests.

get()

page = None

route = '/'

shellbot.routes.wrapper module

class shellbot.routes.wrapper.**Wrapper**(*context=None*, ***kwargs*)

Bases: *shellbot.routes.base.Route*

Calls a function on web request

When the route is requested over the web, the wrapped function is called.

Example:

```
def my_callable(**kwargs):
    ...

route = Wrapper(callable=my_callable, route='/hook')
```

Wrapping is triggered on GET, POST, PUT and DELETE verbs.

callable = None

delete()

get(***kwargs*)

post()

put()

route = None

Module contents

class shellbot.routes.**Route**(*context=None*, ***kwargs*)
Bases: object

Implements one route

delete()

get(***kwargs*)

post()

put()

route = None

class shellbot.routes.**Notifier**(*context=None*, ***kwargs*)
Bases: *shellbot.routes.base.Route*

Notifies a queue on web request

>>>queue = Queue() >>>route = Notifier(route='/notify', queue=queue, notification='hello')

When the route is requested over the web, the notification is pushed to the queue.

>>>queue.get() 'hello'

Notification is triggered on GET, POST, PUT and DELETE verbs.

delete()

get(***kwargs*)

notification = None

notify()

post()

put()

queue = <shellbot.routes.notifier.NoQueue object>

route = '/notify'

class shellbot.routes.**Text**(*context=None*, ***kwargs*)
Bases: *shellbot.routes.base.Route*

Implements a static web page

>>>page = "<html> ... </html>" >>>route = text(route='/index', page=page)

When the route is requested over the web, static content is provided in return.

>>>route.get() "<html> ... </html>"

This class handles only GET requests.

get()

page = None

route = '/'

class shellbot.routes.**Wrapper**(*context=None*, ***kwargs*)

Bases: *shellbot.routes.base.Route*

Calls a function on web request

When the route is requested over the web, the wrapped function is called.

Example:

```
def my_callable(**kwargs):
    ...

route = Wrapper(callable=my_callable, route='/hook')
```

Wrapping is triggered on GET, POST, PUT and DELETE verbs.

callable = None

delete()

get(***kwargs*)

post()

put()

route = None

shellbot.spaces package

Submodules

shellbot.spaces.base module

class shellbot.spaces.base.**Space**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)

Bases: object

Handles a collaborative space

A collaborative space supports multiple channels for interactions between persons and bots.

The life cycle of a space can be described as follows:

1.A space instance is created and configured:

```
>>>my_context = Context(...)
>>>space = Space(context=my_context)
```

2.The space is connected to some back-end API:

```
space.connect()
```

3.Multiple channels can be handled by a single space:

```
channel = space.create(title)

channel = space.get_by_title(title)
channel = space.get_by_id(id)
channel = space.get_by_person(label)

channel.title = 'A new title'
```

```
space.update(channel)

space.delete(id)
```

Channels feature common attributes, yet can be extended to convey specificities of some platforms.

4.Messages can be posted:

```
space.post_message(id, 'Hello, World!')  # for group channels
space.post_message(person, 'Hello, World!')  # for direct messages
```

5.You can add and remove participants to channels:

```
persons = space.list_participant(id)
space.add_participants(id, persons)
space.add_participant(id, person)
space.remove_participants(id, persons)
space.remove_participant(id, person)
```

Multiple modes can be considered for the handling of inbound events from the cloud.

•Asynchronous reception - the back-end API sends updates over a web hook to this object, and messages are pushed to the listening queue.

Example:

```
# link local web server to this space
server.add_route('/hook', space.webhook)

# link cloud service to this local server
space.register('http://my.server/hook')
```

•Background loop - this object pulls the API in a loop, and new messages are pushed to the listening queue.

Example:

```
space.run()
```

DEFAULT_SETTINGS = {'server': {'url': '$SERVER_URL', 'hook': '/hook', 'binding': None, 'port': 8080}, 'space': {'ty

DEFAULT_SPACE_TITLE = u'Collaboration space'

PULL_INTERVAL = 0.05

add_participant(*id*, *person*, *is_moderator=False*)

Adds one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to add
- **is_moderator** (*True or False*) – if this person has special powers on this channel

The underlying platform may, or not, take the optional parameter `is_moderator` into account. The default bahaviour is to discard it, as if the parameter had the value `False`.

This function should be implemented in sub-class. It should not raise exceptions, since this would kill `list_participants()`.

Example:

```
@no_exception
def add_participant(self, id, person):
    self.api.memberships.create(id=id, person=person)
```

add_participants(*id*, *persons=[]*)
Adds multiple participants

Parameters

- **id** (*str*) – the unique id of an existing channel
- **persons** (*list of str*) – e-mail addresses of persons to add

check()
Checks settings

This function should be expanded in sub-class, where necessary.

Example:

```
def check(self):
    self.engine.context.check('space.title',
                              is_mandatory=True)
```

configure(*settings={}*)
Changes settings of the space

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

configured_title()
Returns the title of the space as set in configuration

Returns the configured title, or `Collaboration space`

Return type str

This function should be rewritten in sub-classes if space title does not come from `space.title` parameter.

connect(***kwargs*)
Connects to the back-end API

This function should be expanded in sub-class, where required.

Example:

```
def connect(self, **kwargs):
    self.api = ApiFactory(self.token)
```

create(*title*, ***kwargs*)
Creates a channel

Parameters **title** (*str*) – title of a new channel

Returns Channel

This function returns a representation of the new channel on success, else it should raise an exception.

This function should be implemented in sub-class.

Example:

```
def create(self, title=None, **kwargs):
    handle = self.api.rooms.create(title=title)
    return Channel(handle.attributes)
```

delete(*id*, ***kwargs*)

Deletes a channel

Parameters **id** (*str*) – the unique id of an existing channel

After a call to this function the related channel does not appear anymore in the list of available resources in the chat space. This can be implemented in the back-end either by actual deletion of resources, or by archiving the channel. In the second scenario, the channel could be restored at a later stage if needed.

This function should be implemented in sub-class.

Example:

```
def delete(self, id=id, **kwargs):
    self.api.rooms.delete(id)
```

deregister()

Stops updates from the cloud back-end

This function should be implemented in sub-class.

get_by_id(*id*, ***kwargs*)

Looks for an existing channel by id

Parameters **id** (*str*) – id of the target channel

Returns Channel instance or None

If a channel already exists with this id, a representation of it is returned. Else the value ‘‘None‘‘is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_id(self, id, **kwargs):
    handle = self.api.rooms.lookup(id=id)
    if handle:
        return Channel(handle.attributes)
```

get_by_person(*label*, ***kwargs*)

Looks for an existing private channel with a person

Parameters **label** (*str*) – the display name of the person's account

Returns Channel instance or None

If a channel already exists for this person, a representation of it is returned. Else the value ‘‘None‘‘is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_id(self, id, **kwargs):
    handle = self.api.rooms.lookup(id=id)
    if handle:
        return Channel(handle.attributes)
```

get_by_title(*title=None*, ***kwargs*)

Looks for an existing space by title

Parameters **title** (*str*) – title of the target channel

Returns Channel instance or None

If a channel already exists with this id, a representation of it is returned. Else the value ‘‘None‘‘is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_title(self, title, **kwargs):
    for handle in self.api.rooms.list()
    if handle.title == title:
        return Channel(handle.attributes)
```

list_group_channels(***kwargs*)

Lists available channels

Returns list of Channel

This function should be implemented in sub-class.

Example:

```
def list_group_channels(self, **kwargs):
    for handle in self.api.rooms.list(type='group'):
        yield Channel(handle.attributes)
```

list_messages(*id=None*, *quantity=10*, *stop_id=None*, *up_to=None*, *with_attachment=False*, ***kwargs*)

List messages

Parameters

- **id** (*str*) – the unique id of an existing channel
- **quantity** (*positive integer*) – maximum number of returned messages
- **stop_id** (*str*) – stop on this message id, and do not include it
- **up_to** (*str of ISO date and time*) – stop on this date and time
- **with_attachment** (*True or False*) – to get only messages with some attachments

Returns a list of Message objects

This function fetches messages from one channel, from newest to the oldest. Compared to the bare `walk_messages` function, it brings additional capabilities listed below:

•quantity - limit the maximum number of messages provided

•stop_id - get new messages since the latest we got

•up_to - get messages up a given date and time

•with_attachments - filter messages to retrieve attachments

Example:

```
for message in space.list_messages(id=channel_id):

    do_something_with_message(message)

    if message.url:
        do_something_with_attachment(message.url)
```

list_participants(*id*)
Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

on_init(***kwargs*)
Handles extended initialisation parameters

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, ex_parameter='extra', **kwargs):
    ...
```

on_start()
Reacts when engine is started

This function should be expanded in sub-class, where necessary.

Example:

```
def on_start(self):
    self.load_cache_from_db()
```

on_stop()
reacts when engine is stopped

This function attempts to deregister webhooks, if any. This behaviour can be expanded in sub-class, where necessary.

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)
Posts a message

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – address for a direct message
- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as Markdown or HTML
- **file** (*str*) – URL or local path for an attachment

Example message out of plain text:

```
space.post_message(id=id, text='hello world')
```

Example message with Markdown:

```
space.post_message(id, content='this is a **bold** statement')
```

Example file upload:

```
space.post_message(id, file='./my_file.pdf')
```

Of course, you can combine text with the upload of a file:

```
text = 'This is the presentation that was used for our meeting'
space.post_message(id=id,
                   text=text,
                   file='./my_file.pdf')
```

For direct messages, provide who you want to reach instead of a channel id, like this:

```
space.post_message(person='foo.bar@acme.com', text='hello guy')
```

This function should be implemented in sub-class.

Example:

```
def post_message(self, id, text=None, **kwargs):
    self.api.messages.create(id=id, text=text)
```

pull()

Fetches updates

This function senses most recent items, and pushes them to the listening queue.

This function should be implemented in sub-class.

Example:

```
def pull(self):
    for message in self.api.list_message():
        self.ears.put(message)
```

register(*hook_url*)

Registers to the cloud API for the reception of updates

Parameters **hook_url** (*str*) – web address to be used by cloud service

This function should be implemented in sub-class.

Example:

```
def register(self, hook_url):
    self.api.register(hook_url)
```

remove_participant(*id*, *person*)

Removes one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to delete

This function should be implemented in sub-class. It should not raise exceptions, since this would kill `remove_participants()`.

Example:

```
@no_exception
def remove_participant(self, id, person):
    self.api.memberships.delete(id=id, person=person)
```

remove_participants(*id*, *persons=[]*)

Removes multiple participants

Parameters

- **id** (*str*) – the unique id of an existing channel
- **persons** (*list of str*) – e-mail addresses of persons to delete

run()

Continuously fetches updates

This function senses new items at regular intervals, and pushes them to the listening queue.

Processing is handled in a separate background process, like in the following example:

```
# gets updates in the background
process = space.start()

...

# wait for the end of the process
process.join()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Note: this function should not be invoked if a webhok has been configured.

start(*hook_url=None*)

Starts the update process

Parameters **hook_url** (*str*) – web address to be used by cloud service (optional)

Returns either the process that has been started, or None

If an URL is provided, it is communicated to the back-end API for asynchronous updates.

Else this function starts a separate daemonic process to pull updates in the background.

update(*channel*, ***kwargs*)

Updates an existing channel

Parameters **channel** (Channel) – a representation of the updated channel

This function should raise an exception when the update is not successful.

This function should be implemented in sub-class.

Example:

```
def update(self, channel):
    self.api.rooms.update(channel.attributes)
```

walk_messages(*id=None*, ***kwargs*)

Walk messages

Parameters **id** (*str*) – the unique id of an existing channel

Returns a iterator of Message objects

This function returns messages from a channel, from the newest to the oldest.

This function should be implemented in sub-class

webhook()

Handles updates sent over the internet

This function should use the `request` object to retrieve details of the web transaction.

This function should be implemented in sub-class.

Example:

```
def webhook(self):

    message_id = request.json['data']['id']
    item = self.api.messages.get(messageId=message_id)
    self.ears.put(item._json)
    return "OK"
```

shellbot.spaces.ciscospark module

class shellbot.spaces.ciscospark.**SparkSpace**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)

Bases: *shellbot.spaces.base.Space*

Handles a Cisco Spark room

This is a representation of a chat space hosted at Cisco Spark.

DEFAULT_SETTINGS = {'spark': {'room': '$CHAT_ROOM_TITLE'}, 'server': {'url': '$SERVER_URL', 'hook': '/hook

add_participant(**args*, ***kwargs*)

check()

Checks settings of the space

This function reads key `space` and below, and update the context accordingly:

```
space.configure({'space': {
    'type': 'spark',
    'room': 'My preferred room',
    'participants':
       ['alan.droit@azerty.org', 'bob.nard@support.tv'],
    'team': 'Anchor team',
    'token': '$MY_BOT_TOKEN',
    }})
```

This can also be written in a more compact form:

```
space.configure({'space.room': 'My preferred room',
    'space.token': '$MY_BOT_TOKEN',
    })
```

This function handles following parameters:

- `space.room` - title of the associated Cisco Spark room. This can refer to an environment variable if it starts with `$`, e.g., `$ROOM_TITLE`.
- `space.participants` - list of initial participants. This can be taken from `$CHANNEL_DEFAULT_PARTICIPANTS` from the environment.
- `space.team` - title of a team associated with this room
- `space.token` - private token of the bot, given by Cisco Spark. Instead of putting the real value of the token you are encouraged to use an environment variable instead, e.g., `$MY_BOT_TOKEN`. If `space.token` is not provided, then the function looks for an environment variable `CISCO_SPARK_BOT_TOKEN`.
- `space.audit_token` - token to be used for the audit of chat events. It is recommended that a token of a person is used, so that the visibility is maximised for the proper audit of events. Instead of putting the real value of the token you are encouraged to use an environment variable instead, e.g., `$MY_AUDIT_TOKEN`. If `space.audit_token` is not provided, then the function looks for an environment variable `CISCO_SPARK_AUDIT_TOKEN`.

If a single value is provided for `participants` then it is turned automatically to a list.

Example:

```
>>>space.configure({'space.participants': 'bobby@jah.com'})
>>>space.context.get('space.participants')
['bobby@jah.com']
```

configured_title()

Returns the title of the space as set in configuration

Returns the configured title, or `Collaboration space`

Return type str

This function should be rewritten in sub-classes if space title does not come from `space.room` parameter.

connect(*factory=None*, ***kwargs*)

Connects to the back-end API

Parameters **factory** – an API factory, for test purpose

Type object

If a factory is provided, it is used to get API instances. Else the regular CiscoSparkAPI is invoked instead.

This function loads two instances of Cisco Spark API, one using the bot token, and one using the audit token, if this is available.

create(*title*, *ex_team=None*, ***kwargs*)

Creates a room

Parameters

- **title** (*str*) – title of a new channel
- **ex_team** (*str or object*) – the team attached to this room (optional)

If the parameter `ex_team` is provided, then it can be either a simple name, or a team object featuring an id.

Returns Channel or None

This function returns a representation of the local channel.

delete(*id*, ***kwargs*)
Deletes a room

Parameters **id** (*str*) – the unique id of an existing room

deregister()
Stops inbound flow from Cisco Spark

This function deregisters hooks that it may have created.

Previous webhooks registered with the bot token are all removed before registration. This means that only the most recent instance of the bot will be notified of new invitations.

This function also removes webhooks created with the audit token, if any. So after deregister the audit of individual rooms just stops.

download_attachment(*url*, *token=None*)
Copies a shared document locally

get_attachment(*url*, *token=None*, *response=None*)
Retrieves a document attached to a room

Returns a stream of BytesIO

Return type BytesIO

get_by_id(*id*, ***kwargs*)
Looks for an existing room by id

Parameters **id** (*str*) – identifier of the target room

Returns Channel instance or None

get_by_person(*label*, ***kwargs*)
Looks for an existing private room with a person

Parameters **label** (*str*) – the display name of the person's account

Returns Channel instance or None

If a channel already exists for this person, a representation of it is returned. Else the value ‘‘None‘‘is returned.

get_by_title(*title*, ***kwargs*)
Looks for an existing room by name

Parameters **title** (*str*) – title of the target room

Returns Channel instance or None

Note: This function looks only into group rooms. To get a direct room use `get_by_person()` instead.

get_team(*name*)
Gets a team by name

Parameters **name** (*str*) – name of the target team

Returns attributes of the team

Return type Team or None

>>>print(space.get_team(“Hello World”)) Team({

“id” : “Y2lzY29zcGFyazovL3VzL1RFQU0Yy0xMWU2LWE5ZDgtMjExYTBkYzc5NzY5”, “name” : “Hello World”, “created” : “2015-10-18T14:26:16+00:00”

})

list_group_channels(*quantity=10*, ***kwargs*)
Lists available channels

Parameters **quantity** (*positive integer*) – maximum quantity of channels to return

Returns list of Channel

list_participants(*id*)
Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

name_attachment(*url*, *token=None*, *response=None*)
Retrieves a document attached to a room

on_connect()
Retrieves attributes of this bot

This function queries the Cisco Spark API to remember the id of this bot. This is used afterwards to filter inbound messages to the shell.

on_init(*token=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **token** (*str*) – bot authentication token for the Cisco Spark API

Example:

```
space = SparkSpace(context=context)
```

on_join(*item*, *queue=None*)
Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Example item received on memberships:create:

```
{
    'isMonitor': False,
    'created': '2017-05-31T21:25:30.424Z',
    'personId': 'Y2lzY29zcGFyazovL3VRiMTAtODZkYy02YzU0Yjg5ODA5N2U',
    'isModerator': False,
    'personOrgId': 'Y2lzY29zcGFyazovL3V0FOSVpBVElPTi9jb25zdW1lcg',
    'personDisplayName': 'foo.bar@acme.com',
    'personEmail': 'foo.bar@acme.com',
    'roomId': 'Y2lzY29zcGFyazovL3VzL1JP3LTk5MDAtMDU5MDI2YjBiNDUz',
    'id': 'Y2lzY29zcGFyazovL3VzDctMTFlNy05OTAwLTA1OTAyNmIwYjQ1Mw'
}
```

This function prepares a Join and push it to the provided queue.

- `type` is set to `join`
- `actor_id` is a copy of `personId`

- `actor_address` is a copy of `personEmail`
- `actor_label` is a copy of `personDisplayName`
- `stamp` is a copy of `created`

on_leave(*item*, *queue=None*)

Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Example item received on memberships:delete:

```
{
    'isMonitor': False,
    'created': '2017-05-31T21:25:30.424Z',
    'personId': 'Y2lzY29zcGFyazovL3VRiMTAtODZkYy02YzU0Yjg5ODA5N2U',
    'isModerator': False,
    'personOrgId': 'Y2lzY29zcGFyazovL3V0FOSVpBVElPTi9jb25zdW1lcg',
    'personDisplayName': 'foo.bar@acme.com',
    'personEmail': 'foo.bar@acme.com',
    'roomId': 'Y2lzY29zcGFyazovL3VzL1JP3LTk5MDAtMDU5MDI2YjBiNDUz',
    'id': 'Y2lzY29zcGFyazovL3VzDctMTFlNy05OTAwLTA1OTAyNmIwYjQ1Mw'
}
```

This function prepares a Leave and push it to the provided queue.

- `type` is set to `leave`
- `actor_id` is a copy of `personId`
- `actor_address` is a copy of `personEmail`
- `actor_label` is a copy of `personDisplayName`
- `stamp` is a copy of `created`

on_message(*item*, *queue=None*)

Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Returns a Message

This function prepares a Message and push it to the provided queue.

This function adds following keys to messages so that a neutral format can be used with the listener:

- `type` is set to `message`
- `content` is a copy of `html`
- `from_id` is a copy of `personId`
- `from_label` is a copy of `personEmail`
- `is_direct` if the message is coming from 1:1 room
- `mentioned_ids` is a copy of `mentionedPeople`

- `channel_id` is a copy of `roomId`
- `stamp` is a copy of `created`

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)

Posts a message to a Cisco Spark room

Parameters

- **id** (*str*) – the unique id of an existing room
- **person** (*str*) – address for a direct message
- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as Markdown or HTML
- **file** (*str*) – URL or local path for an attachment

Example message out of plain text:

```
space.post_message(id=id, text='hello world')
```

Example message with Markdown:

```
space.post_message(id, content='this is a **bold** statement')
```

Example file upload:

```
space.post_message(id, file='./my_file.pdf')
```

Of course, you can combine text with the upload of a file:

```
text = 'This is the presentation that was used for our meeting'
space.post_message(id=id,
                   text=text,
                   file='./my_file.pdf')
```

For direct messages, provide who you want to reach instead of a channel id, like this:

```
space.post_message(person='foo.bar@acme.com', text='hello guy')
```

pull()

Fetches events from Cisco Spark

This function senses most recent items, and pushes them to a processing queue.

register(*hook_url*)

Connects in the background to Cisco Spark inbound events

Parameters **webhook** (*str*) – web address to be used by Cisco Spark service

This function registers the provided hook multiple times, so as to receive mutiple kind of updates:

- The bot is invited to a room, or kicked out of it. People are joining or leaving: webhook name = shellbot-memberships resource = memberships, event = all, registered with bot token
- Messages are sent, maybe with some files: webhook name = shellbot-messages resource = messages, event = created, registered with bot token
- Messages sent, maybe with some files, for audit purpose: webhook name = shellbot-audit resource = messages, event = created, registered with audit token

Previous webhooks registered with the bot token are all removed before registration. This means that only the most recent instance of the bot will be notified of new invitations.

remove_participant(**args*, ***kwargs*)

update(*channel*, ***kwargs*)
Updates an existing room

Parameters **channel** (Channel) – a representation of the updated room

This function can change the title of a room.

For example, change the title from a bot instance:

```
bot.channel.title = "A new title"
bot.space.update(bot.channel)
```

walk_messages(*id=None*, ***kwargs*)
Walk messages from a Cisco Spark room

Parameters **id** (*str*) – the unique id of an existing room

Returns an iterator of Message objects

webhook(*item=None*)
Processes the flow of events from Cisco Spark

Parameters **item** (*dict*) – if provided, do not invoke the `request` object

This function is called from far far away, over the Internet, most of the time. Or it is called locally, from test environment, when an item is provided.

The structure of the provided item should be identical to those of updates sent by Cisco Spark.

Example event on message creation:

```
{
    "resource": "messages",
    "event": "created",
    "data": { "id": "...." },
    "name": "shellbot-audit"
}
```

shellbot.spaces.ciscospark.**no_exception**(*function*, *return_value=None*)
Stops the propagation of exceptions

Parameters **return_value** – Returned by the decorated function on exception

This decorator is a convenient approach for silently discarding exceptions.

#wip – this should be moved in a general-purpose module of shellbot

Example:

```
@no_exception(return_value=[])
def list_items():
    ...  # if an exception is raised here, an empty list is returned
```

shellbot.spaces.ciscospark.**retry**(*give_up='Unable to request Cisco Spark API'*, *silent=False*, *delays=(0.1, 1, 5)*, *skipped=(401, 403, 404, 409)*)
Improves a call to Cisco Spark API

Parameters

- **give_up** (*str*) – message to log on final failure

- **silent** (*bool*) – if exceptions should be masked as much as possible
- **delays** (*a list of positive numbers*) – time to wait between repetitions
- **skipped** (*a list of web status codes*) – do not retry for these status codes

This decorator compensates for common transient communication issues with the Cisco Spark platform in the cloud.

Example:

```
@retry(give_up="Unable to get information on this bot")
def api_call():
    return self.api.people.me()

me = api_call()
```

credit: http://code.activestate.com/recipes/580745-retry-decorator-in-python/

shellbot.spaces.local module

class shellbot.spaces.local.**LocalSpace**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)
Bases: *shellbot.spaces.base.Space*

Handles chat locally

This class allows developers to test their commands interface locally, without the need for a real API back-end.

If a list of commands is provided as input, then the space will consume all of them and then it will stop. All kinds of automated tests and scenarios can be build with this approach.

Example of automated interaction with some commands:

```
engine = Engine(command=Hello(), type='local')
engine.space.push(['help', 'hello', 'help help'])

engine.configure()
engine.run()
```

If no input is provided, then the space provides a command-line interface so that you can play interactively with your bot. This setup is handy since it does not require access to a real chat back-end.

DEFAULT_PROMPT = u'> '

add_participant(*id*, *person*, *is_moderator=False*)
Adds one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to add
- **is_moderator** (*True or False*) – if this person has special powers on this channel

check()
Check settings

This function reads key `local` and below, and update the context accordingly.

This function also selects the right input for this local space. If some content has been provided during initialisation, it is used to simulate user input. Else stdin is read one line at a time.

create (*title*, ***kwargs*)
Creates a channel

Parameters **title** (*str*) – title of a new channel

Returns Channel

This function returns a representation of the local channel.

delete (*id*, ***kwargs*)
Deletes a channel

Parameters **id** (*str*) – the unique id of an existing channel

get_by_id (*id*, ***kwargs*)
Looks for an existing channel by id

Parameters **id** (*str*) – identifier of the target channel

Returns Channel instance or None

get_by_title (*title*, ***kwargs*)
Looks for an existing channel by title

Parameters **title** (*str*) – title of the target channel

Returns Channel instance or None

list_group_channels (***kwargs*)
Lists available channels

Returns list of Channel

list_participants (*id*)
Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

on_init (*input=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **input** (*str or list of str*) – Lines of text to be submitted to the chat

Example:

```
space = LocalSpace(input='hello world')
```

Here we create a new local space, and simulate a user typing 'hello world' in the chat space.

on_message (*item*, *queue*)
Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue

This function prepares a Message and push it to the provided queue.

on_start ()
Adds processing on engine start

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)
Posts a message

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – address for a direct message
- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as MArkdown or HTML
- **file** (*str*) – URL or local path for an attachment

pull()
Fetches updates

This function senses most recent item, and pushes it to the listening queue.

push(*input*)
Adds more input to this space

Parameters **input** (*str or list of str*) – Simulated user input

This function is used to simulate input user to the bot.

remove_participant(*id*, *person*)
Removes one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to remove

update(*channel*, ***kwargs*)
Updates an existing channel

Parameters **channel** (Channel) – a representation of the updated channel

walk_messages(*id=None*, ***kwargs*)
Walk messages

Parameters **id** (*str*) – the unique id of an existing channel

Returns a iterator of Message objects

This function returns messages from a channel, from the newest to the oldest.

Module contents

class shellbot.spaces.**SpaceFactory**
Bases: object

Builds a space from configuration

Example:

```
my_context = Context(settings={
    'space': {
        'type': "spark",
        'room': 'My preferred room',
        'participants':
            ['alan.droit@azerty.org', 'bob.nard@support.tv'],
```

```
        'team': 'Anchor team',
        'token': 'hkNWEtMJNkODk3ZDZLOGQ0OVGlZWU1NmYtyY',
        'fuzzy_token': '$MY_FUZZY_SPARK_TOKEN',
    }
})

space = SpaceFactory.build(context=my_context)
```

classmethod build(*context*, ***kwargs*)

Builds an instance based on provided configuration

Parameters **context** (Context) – configuration to be used

Returns a ready-to-use space

Return type *Space*

This function "senses" for a type in the context itself, then provides with an instantiated object of this type.

A ValueError is raised when no type can be identified.

classmethod get(*type*, ***kwargs*)

Loads a space by type

Parameters **type** (*str*) – the required space

Returns a space instance

This function seeks for a suitable space class in the library, and returns an instance of it.

Example:

```
space = SpaceFactory.get('spark', ex_token='123')
```

A ValueError is raised if the type is unknown.

classmethod sense(*context*)

Detects type from configuration

Parameters **context** (Context) – configuration to be analyzed

Returns a guessed type

Return type str

Example:

```
type = SpaceFactory.sense(context)
```

A ValueError is raised if no type could be identified.

types = {'spark': <class 'shellbot.spaces.ciscospark.SparkSpace'>, 'local': <class 'shellbot.spaces.local.LocalSpace'>, 'sp

class shellbot.spaces.**Space**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)

Bases: object

Handles a collaborative space

A collaborative space supports multiple channels for interactions between persons and bots.

The life cycle of a space can be described as follows:

1. A space instance is created and configured:

```
>>>my_context = Context(...)
>>>space = Space(context=my_context)
```

2.The space is connected to some back-end API:

```
space.connect()
```

3.Multiple channels can be handled by a single space:

```
channel = space.create(title)

channel = space.get_by_title(title)
channel = space.get_by_id(id)
channel = space.get_by_person(label)

channel.title = 'A new title'
space.update(channel)

space.delete(id)
```

Channels feature common attributes, yet can be extended to convey specificities of some platforms.

4.Messages can be posted:

```
space.post_message(id, 'Hello, World!')  # for group channels
space.post_message(person, 'Hello, World!')  # for direct messages
```

5.You can add and remove participants to channels:

```
persons = space.list_participant(id)
space.add_participants(id, persons)
space.add_participant(id, person)
space.remove_participants(id, persons)
space.remove_participant(id, person)
```

Multiple modes can be considered for the handling of inbound events from the cloud.

•Asynchronous reception - the back-end API sends updates over a web hook to this object, and messages are pushed to the listening queue.

Example:

```
# link local web server to this space
server.add_route('/hook', space.webhook)

# link cloud service to this local server
space.register('http://my.server/hook')
```

•Background loop - this object pulls the API in a loop, and new messages are pushed to the listening queue.

Example:

```
space.run()
```

DEFAULT_SETTINGS = {'server': {'url': '$SERVER_URL', 'hook': '/hook', 'binding': None, 'port': 8080}, 'space': {'ty

DEFAULT_SPACE_TITLE = u'Collaboration space'

PULL_INTERVAL = 0.05

add_participant(*id*, *person*, *is_moderator=False*)
Adds one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to add
- **is_moderator** (*True or False*) – if this person has special powers on this channel

The underlying platform may, or not, take the optional parameter `is_moderator` into account. The default bahaviour is to discard it, as if the parameter had the value `False`.

This function should be implemented in sub-class. It should not raise exceptions, since this would kill `list_participants()`.

Example:

```
@no_exception
def add_participant(self, id, person):
    self.api.memberships.create(id=id, person=person)
```

add_participants(*id*, *persons=[]*)
Adds multiple participants

Parameters

- **id** (*str*) – the unique id of an existing channel
- **persons** (*list of str*) – e-mail addresses of persons to add

check()
Checks settings

This function should be expanded in sub-class, where necessary.

Example:

```
def check(self):
    self.engine.context.check('space.title',
                              is_mandatory=True)
```

configure(*settings={}*)
Changes settings of the space

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

configured_title()
Returns the title of the space as set in configuration

Returns the configured title, or `Collaboration space`

Return type str

This function should be rewritten in sub-classes if space title does not come from `space.title` parameter.

connect(***kwargs*)
Connects to the back-end API

This function should be expanded in sub-class, where required.

Example:

```
def connect(self, **kwargs):
    self.api = ApiFactory(self.token)
```

create(*title*, ***kwargs*)

Creates a channel

Parameters **title** (*str*) – title of a new channel

Returns Channel

This function returns a representation of the new channel on success, else it should raise an exception.

This function should be implemented in sub-class.

Example:

```
def create(self, title=None, **kwargs):
    handle = self.api.rooms.create(title=title)
    return Channel(handle.attributes)
```

delete(*id*, ***kwargs*)

Deletes a channel

Parameters **id** (*str*) – the unique id of an existing channel

After a call to this function the related channel does not appear anymore in the list of available resources in the chat space. This can be implemented in the back-end either by actual deletion of resources, or by archiving the channel. In the second scenario, the channel could be restored at a later stage if needed.

This function should be implemented in sub-class.

Example:

```
def delete(self, id=id, **kwargs):
    self.api.rooms.delete(id)
```

deregister()

Stops updates from the cloud back-end

This function should be implemented in sub-class.

get_by_id(*id*, ***kwargs*)

Looks for an existing channel by id

Parameters **id** (*str*) – id of the target channel

Returns Channel instance or None

If a channel already exists with this id, a representation of it is returned. Else the value ‘‘None‘‘is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_id(self, id, **kwargs):
    handle = self.api.rooms.lookup(id=id)
    if handle:
        return Channel(handle.attributes)
```

get_by_person(*label*, ***kwargs*)

Looks for an existing private channel with a person

Parameters **label** (*str*) – the display name of the person's account

Returns Channel instance or None

If a channel already exists for this person, a representation of it is returned. Else the value ''None'' is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_id(self, id, **kwargs):
    handle = self.api.rooms.lookup(id=id)
    if handle:
        return Channel(handle.attributes)
```

get_by_title(*title=None*, ***kwargs*)

Looks for an existing space by title

Parameters **title** (*str*) – title of the target channel

Returns Channel instance or None

If a channel already exists with this id, a representation of it is returned. Else the value ''None'' is returned.

This function should be implemented in sub-class.

Example:

```
def get_by_title(self, title, **kwargs):
    for handle in self.api.rooms.list()
    if handle.title == title:
        return Channel(handle.attributes)
```

list_group_channels(***kwargs*)

Lists available channels

Returns list of Channel

This function should be implemented in sub-class.

Example:

```
def list_group_channels(self, **kwargs):
    for handle in self.api.rooms.list(type='group'):
        yield Channel(handle.attributes)
```

list_messages(*id=None*, *quantity=10*, *stop_id=None*, *up_to=None*, *with_attachment=False*, ***kwargs*)

List messages

Parameters

- **id** (*str*) – the unique id of an existing channel
- **quantity** (*positive integer*) – maximum number of returned messages
- **stop_id** (*str*) – stop on this message id, and do not include it
- **up_to** (*str of ISO date and time*) – stop on this date and time
- **with_attachment** (*True or False*) – to get only messages with some attachments

Returns a list of Message objects

This function fetches messages from one channel, from newest to the oldest. Compared to the bare `walk_messages` function, it brings additional capabilities listed below:

•quantity - limit the maximum number of messages provided

•stop_id - get new messages since the latest we got

•up_to - get messages up a given date and time

•with_attachments - filter messages to retrieve attachments

Example:

```
for message in space.list_messages(id=channel_id):

    do_something_with_message(message)

    if message.url:
        do_something_with_attachment(message.url)
```

list_participants(*id*)

Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

on_init(***kwargs*)

Handles extended initialisation parameters

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, ex_parameter='extra', **kwargs):
    ...
```

on_start()

Reacts when engine is started

This function should be expanded in sub-class, where necessary.

Example:

```
def on_start(self):
    self.load_cache_from_db()
```

on_stop()

reacts when engine is stopped

This function attempts to deregister webhooks, if any. This behaviour can be expanded in sub-class, where necessary.

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)

Posts a message

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – address for a direct message

- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as Markdown or HTML
- **file** (*str*) – URL or local path for an attachment

Example message out of plain text:

```
space.post_message(id=id, text='hello world')
```

Example message with Markdown:

```
space.post_message(id, content='this is a **bold** statement')
```

Example file upload:

```
space.post_message(id, file='./my_file.pdf')
```

Of course, you can combine text with the upload of a file:

```
text = 'This is the presentation that was used for our meeting'
space.post_message(id=id,
                   text=text,
                   file='./my_file.pdf')
```

For direct messages, provide who you want to reach instead of a channel id, like this:

```
space.post_message(person='foo.bar@acme.com', text='hello guy')
```

This function should be implemented in sub-class.

Example:

```
def post_message(self, id, text=None, **kwargs):
    self.api.messages.create(id=id, text=text)
```

pull()

Fetches updates

This function senses most recent items, and pushes them to the listening queue.

This function should be implemented in sub-class.

Example:

```
def pull(self):
    for message in self.api.list_message():
        self.ears.put(message)
```

register(*hook_url*)

Registers to the cloud API for the reception of updates

Parameters **hook_url** (*str*) – web address to be used by cloud service

This function should be implemented in sub-class.

Example:

```
def register(self, hook_url):
    self.api.register(hook_url)
```

remove_participant(*id*, *person*)
Removes one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to delete

This function should be implemented in sub-class. It should not raise exceptions, since this would kill `remove_participants()`.

Example:

```
@no_exception
def remove_participant(self, id, person):
    self.api.memberships.delete(id=id, person=person)
```

remove_participants(*id*, *persons=[]*)
Removes multiple participants

Parameters

- **id** (*str*) – the unique id of an existing channel
- **persons** (*list of str*) – e-mail addresses of persons to delete

run()
Continuously fetches updates

This function senses new items at regular intervals, and pushes them to the listening queue.

Processing is handled in a separate background process, like in the following example:

```
# gets updates in the background
process = space.start()

...

# wait for the end of the process
process.join()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Note: this function should not be invoked if a webhok has been configured.

start(*hook_url=None*)
Starts the update process

Parameters **hook_url** (*str*) – web address to be used by cloud service (optional)

Returns either the process that has been started, or None

If an URL is provided, it is communicated to the back-end API for asynchronous updates.

Else this function starts a separate daemonic process to pull updates in the background.

update(*channel*, ***kwargs*)
Updates an existing channel

Parameters **channel** (Channel) – a representation of the updated channel

This function should raise an exception when the update is not successful.

This function should be implemented in sub-class.

Example:

```
def update(self, channel):
    self.api.rooms.update(channel.attributes)
```

walk_messages(*id=None*, ***kwargs*)

Walk messages

Parameters **id** (*str*) – the unique id of an existing channel

Returns a iterator of Message objects

This function returns messages from a channel, from the newest to the oldest.

This function should be implemented in sub-class

webhook()

Handles updates sent over the internet

This function should use the `request` object to retrieve details of the web transaction.

This function should be implemented in sub-class.

Example:

```
def webhook(self):

    message_id = request.json['data']['id']
    item = self.api.messages.get(messageId=message_id)
    self.ears.put(item._json)
    return "OK"
```

class shellbot.spaces.**LocalSpace**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)

Bases: *shellbot.spaces.base.Space*

Handles chat locally

This class allows developers to test their commands interface locally, without the need for a real API back-end.

If a list of commands is provided as input, then the space will consume all of them and then it will stop. All kinds of automated tests and scenarios can be build with this approach.

Example of automated interaction with some commands:

```
engine = Engine(command=Hello(), type='local')
engine.space.push(['help', 'hello', 'help help'])

engine.configure()
engine.run()
```

If no input is provided, then the space provides a command-line interface so that you can play interactively with your bot. This setup is handy since it does not require access to a real chat back-end.

DEFAULT_PROMPT = u'> '

add_participant(*id*, *person*, *is_moderator=False*)

Adds one participant

Parameters

- **id** (*str*) – the unique id of an existing channel

- **person** (*str*) – e-mail address of the person to add
- **is_moderator** (*True or False*) – if this person has special powers on this channel

check()
Check settings

This function reads key `local` and below, and update the context accordingly.

This function also selects the right input for this local space. If some content has been provided during initialisation, it is used to simulate user input. Else stdin is read one line at a time.

create(*title*, ***kwargs*)
Creates a channel

Parameters **title** (*str*) – title of a new channel

Returns Channel

This function returns a representation of the local channel.

delete(*id*, ***kwargs*)
Deletes a channel

Parameters **id** (*str*) – the unique id of an existing channel

get_by_id(*id*, ***kwargs*)
Looks for an existing channel by id

Parameters **id** (*str*) – identifier of the target channel

Returns Channel instance or None

get_by_title(*title*, ***kwargs*)
Looks for an existing channel by title

Parameters **title** (*str*) – title of the target channel

Returns Channel instance or None

list_group_channels(***kwargs*)
Lists available channels

Returns list of Channel

list_participants(*id*)
Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

on_init(*input=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **input** (*str or list of str*) – Lines of text to be submitted to the chat

Example:

```
space = LocalSpace(input='hello world')
```

Here we create a new local space, and simulate a user typing 'hello world' in the chat space.

on_message(*item*, *queue*)
Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue

This function prepares a Message and push it to the provided queue.

on_start()
Adds processing on engine start

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)
Posts a message

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – address for a direct message
- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as MArkdown or HTML
- **file** (*str*) – URL or local path for an attachment

pull()
Fetches updates

This function senses most recent item, and pushes it to the listening queue.

push(*input*)
Adds more input to this space

Parameters **input** (*str or list of str*) – Simulated user input

This function is used to simulate input user to the bot.

remove_participant(*id*, *person*)
Removes one participant

Parameters

- **id** (*str*) – the unique id of an existing channel
- **person** (*str*) – e-mail address of the person to remove

update(*channel*, ***kwargs*)
Updates an existing channel

Parameters **channel** (Channel) – a representation of the updated channel

walk_messages(*id=None*, ***kwargs*)
Walk messages

Parameters **id** (*str*) – the unique id of an existing channel

Returns a iterator of Message objects

This function returns messages from a channel, from the newest to the oldest.

class shellbot.spaces.**SparkSpace**(*context=None*, *ears=None*, *fan=None*, ***kwargs*)
Bases: *shellbot.spaces.base.Space*

Handles a Cisco Spark room

This is a representation of a chat space hosted at Cisco Spark.

DEFAULT_SETTINGS = {'spark': {'room': '$CHAT_ROOM_TITLE'}, 'server': {'url': '$SERVER_URL', 'hook': '/hook

add_participant(**args*, ***kwargs*)

check()

Checks settings of the space

This function reads key `space` and below, and update the context accordingly:

```
space.configure({'space': {
    'type': 'spark',
    'room': 'My preferred room',
    'participants':
       ['alan.droit@azerty.org', 'bob.nard@support.tv'],
    'team': 'Anchor team',
    'token': '$MY_BOT_TOKEN',
    }})
```

This can also be written in a more compact form:

```
space.configure({'space.room': 'My preferred room',
    'space.token': '$MY_BOT_TOKEN',
    })
```

This function handles following parameters:

- `space.room` - title of the associated Cisco Spark room. This can refer to an environment variable if it starts with `$`, e.g., `$ROOM_TITLE`.
- `space.participants` - list of initial participants. This can be taken from `$CHANNEL_DEFAULT_PARTICIPANTS` from the environment.
- `space.team` - title of a team associated with this room
- `space.token` - private token of the bot, given by Cisco Spark. Instead of putting the real value of the token you are encouraged to use an environment variable instead, e.g., `$MY_BOT_TOKEN`. If `space.token` is not provided, then the function looks for an environment variable `CISCO_SPARK_BOT_TOKEN`.
- `space.audit_token` - token to be used for the audit of chat events. It is recommended that a token of a person is used, so that the visibility is maximised for the proper audit of events. Instead of putting the real value of the token you are encouraged to use an environment variable instead, e.g., `$MY_AUDIT_TOKEN`. If `space.audit_token` is not provided, then the function looks for an environment variable `CISCO_SPARK_AUDIT_TOKEN`.

If a single value is provided for `participants` then it is turned automatically to a list.

Example:

```
>>>space.configure({'space.participants': 'bobby@jah.com'})
>>>space.context.get('space.participants')
['bobby@jah.com']
```

configured_title()

Returns the title of the space as set in configuration

Returns the configured title, or `Collaboration space`

Return type str

This function should be rewritten in sub-classes if space title does not come from `space.room` parameter.

connect(*factory=None*, ***kwargs*)
Connects to the back-end API

Parameters **factory** – an API factory, for test purpose

Type object

If a factory is provided, it is used to get API instances. Else the regular CiscoSparkAPI is invoked instead.

This function loads two instances of Cisco Spark API, one using the bot token, and one using the audit token, if this is available.

create(*title*, *ex_team=None*, ***kwargs*)
Creates a room

Parameters

- **title** (*str*) – title of a new channel
- **ex_team** (*str or object*) – the team attached to this room (optional)

If the parameter `ex_team` is provided, then it can be either a simple name, or a team object featuring an id.

Returns Channel or None

This function returns a representation of the local channel.

delete(*id*, ***kwargs*)
Deletes a room

Parameters **id** (*str*) – the unique id of an existing room

deregister()
Stops inbound flow from Cisco Spark

This function deregisters hooks that it may have created.

Previous webhooks registered with the bot token are all removed before registration. This means that only the most recent instance of the bot will be notified of new invitations.

This function also removes webhooks created with the audit token, if any. So after deregister the audit of individual rooms just stops.

download_attachment(*url*, *token=None*)
Copies a shared document locally

get_attachment(*url*, *token=None*, *response=None*)
Retrieves a document attached to a room

Returns a stream of BytesIO

Return type BytesIO

get_by_id(*id*, ***kwargs*)
Looks for an existing room by id

Parameters **id** (*str*) – identifier of the target room

Returns Channel instance or None

get_by_person(*label*, ***kwargs*)
Looks for an existing private room with a person

Parameters **label** (*str*) – the display name of the person's account

Returns Channel instance or None

If a channel already exists for this person, a representation of it is returned. Else the value ‘‘None‘‘is returned.

get_by_title(*title*, ***kwargs*)
Looks for an existing room by name

Parameters **title** (*str*) – title of the target room

Returns Channel instance or None

Note: This function looks only into group rooms. To get a direct room use `get_by_person()` instead.

get_team(*name*)
Gets a team by name

Parameters **name** (*str*) – name of the target team

Returns attributes of the team

Return type Team or None

>>>print(space.get_team("Hello World")) Team({

"id" : "Y2lzY29zcGFyazovL3VzL1RFQU0Yy0xMWU2LWE5ZDgtMjExYTBkYzc5NzY5", "name" : "Hello World", "created" : "2015-10-18T14:26:16+00:00"

})

list_group_channels(*quantity=10*, ***kwargs*)
Lists available channels

Parameters **quantity** (*positive integer*) – maximum quantity of channels to return

Returns list of Channel

list_participants(*id*)
Lists participants to a channel

Parameters **id** (*str*) – the unique id of an existing channel

Returns a list of persons

Return type list of str

Note: this function returns all participants, except the bot itself.

name_attachment(*url*, *token=None*, *response=None*)
Retrieves a document attached to a room

on_connect()
Retrieves attributes of this bot

This function queries the Cisco Spark API to remember the id of this bot. This is used afterwards to filter inbound messages to the shell.

on_init(*token=None*, ***kwargs*)
Handles extended initialisation parameters

Parameters **token** (*str*) – bot authentication token for the Cisco Spark API

Example:

```
space = SparkSpace(context=context)
```

on_join (*item*, *queue=None*)

Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Example item received on memberships:create:

```
{
    'isMonitor': False,
    'created': '2017-05-31T21:25:30.424Z',
    'personId': 'Y2lzY29zcGFyazovL3VRiMTAtODZkYy02YzU0Yjg5ODA5N2U',
    'isModerator': False,
    'personOrgId': 'Y2lzY29zcGFyazovL3V0FOSVpBVElPTi9jb25zdW1lcg',
    'personDisplayName': 'foo.bar@acme.com',
    'personEmail': 'foo.bar@acme.com',
    'roomId': 'Y2lzY29zcGFyazovL3VzL1JP3LTk5MDAtMDU5MDI2YjBiNDUz',
    'id': 'Y2lzY29zcGFyazovL3VzDctMTFlNy05OTAwLTA1OTAyNmIwYjQ1Mw'
}
```

This function prepares a Join and push it to the provided queue.

- `type` is set to `join`
- `actor_id` is a copy of `personId`
- `actor_address` is a copy of `personEmail`
- `actor_label` is a copy of `personDisplayName`
- `stamp` is a copy of `created`

on_leave (*item*, *queue=None*)

Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Example item received on memberships:delete:

```
{
    'isMonitor': False,
    'created': '2017-05-31T21:25:30.424Z',
    'personId': 'Y2lzY29zcGFyazovL3VRiMTAtODZkYy02YzU0Yjg5ODA5N2U',
    'isModerator': False,
    'personOrgId': 'Y2lzY29zcGFyazovL3V0FOSVpBVElPTi9jb25zdW1lcg',
    'personDisplayName': 'foo.bar@acme.com',
    'personEmail': 'foo.bar@acme.com',
    'roomId': 'Y2lzY29zcGFyazovL3VzL1JP3LTk5MDAtMDU5MDI2YjBiNDUz',
    'id': 'Y2lzY29zcGFyazovL3VzDctMTFlNy05OTAwLTA1OTAyNmIwYjQ1Mw'
}
```

This function prepares a Leave and push it to the provided queue.

- `type` is set to `leave`

- `actor_id` is a copy of `personId`
- `actor_address` is a copy of `personEmail`
- `actor_label` is a copy of `personDisplayName`
- `stamp` is a copy of `created`

on_message(*item*, *queue=None*)

Normalizes message for the listener

Parameters

- **item** (*dict*) – attributes of the inbound message
- **queue** (*Queue*) – the processing queue (optional)

Returns a Message

This function prepares a Message and push it to the provided queue.

This function adds following keys to messages so that a neutral format can be used with the listener:

- `type` is set to `message`
- `content` is a copy of `html`
- `from_id` is a copy of `personId`
- `from_label` is a copy of `personEmail`
- `is_direct` if the message is coming from 1:1 room
- `mentioned_ids` is a copy of `mentionedPeople`
- `channel_id` is a copy of `roomId`
- `stamp` is a copy of `created`

post_message(*id=None*, *text=None*, *content=None*, *file=None*, *person=None*, ***kwargs*)

Posts a message to a Cisco Spark room

Parameters

- **id** (*str*) – the unique id of an existing room
- **person** (*str*) – address for a direct message
- **text** (*str*) – message in plain text
- **content** (*str*) – rich format, such as Markdown or HTML
- **file** (*str*) – URL or local path for an attachment

Example message out of plain text:

```
space.post_message(id=id, text='hello world')
```

Example message with Markdown:

```
space.post_message(id, content='this is a **bold** statement')
```

Example file upload:

```
space.post_message(id, file='./my_file.pdf')
```

Of course, you can combine text with the upload of a file:

```
text = 'This is the presentation that was used for our meeting'
space.post_message(id=id,
                   text=text,
                   file='./my_file.pdf')
```

For direct messages, provide who you want to reach instead of a channel id, like this:

```
space.post_message(person='foo.bar@acme.com', text='hello guy')
```

pull()
Fetches events from Cisco Spark

This function senses most recent items, and pushes them to a processing queue.

register(*hook_url*)
Connects in the background to Cisco Spark inbound events

Parameters **webhook** (*str*) – web address to be used by Cisco Spark service

This function registers the provided hook multiple times, so as to receive mutiple kind of updates:

- The bot is invited to a room, or kicked out of it. People are joining or leaving: webhook name = shellbot-memberships resource = memberships, event = all, registered with bot token
- Messages are sent, maybe with some files: webhook name = shellbot-messages resource = messages, event = created, registered with bot token
- Messages sent, maybe with some files, for audit purpose: webhook name = shellbot-audit resource = messages, event = created, registered with audit token

Previous webhooks registered with the bot token are all removed before registration. This means that only the most recent instance of the bot will be notified of new invitations.

remove_participant(**args*, ***kwargs*)

update(*channel*, ***kwargs*)
Updates an existing room

Parameters **channel** (Channel) – a representation of the updated room

This function can change the title of a room.

For example, change the title from a bot instance:

```
bot.channel.title = "A new title"
bot.space.update(bot.channel)
```

walk_messages(*id=None*, ***kwargs*)
Walk messages from a Cisco Spark room

Parameters **id** (*str*) – the unique id of an existing room

Returns an iterator of Message objects

webhook(*item=None*)
Processes the flow of events from Cisco Spark

Parameters **item** (*dict*) – if provided, do not invoke the `request` object

This function is called from far far away, over the Internet, most of the time. Or it is called locally, from test environment, when an item is provided.

The structure of the provided item should be identical to those of updates sent by Cisco Spark.

Example event on message creation:

```
{
    "resource": "messages",
    "event": "created",
    "data": { "id": "...." },
    "name": "shellbot-audit"
}
```

shellbot.stores package

Submodules

shellbot.stores.base module

class shellbot.stores.base.**Store**(*context=None*, ***kwargs*)

Bases: object

Stores data for one space

This is a key-value store, that supports concurrency across multiple processes.

Configuration of the storage engine is coming from settings of the overall bot.

Example:

```
store = Store(context=my_context)
```

Normally a store is related to one single space. For this, you can use the function `bond()` to set the space unique id.

Example:

```
store.bond(id=space.id)
```

Once this is done, the store can be used to remember and to recall values.

Example:

```
store.remember('gauge', gauge)
...
gauge = store.recall('gauge')
```

append(*key*, *item*)

Appends an item to a list

Parameters

- **key** (*str*) – name of the list
- **item** (*any serializable type is accepted*) – a new item to append

Example:

```
>>>store.append('names', 'Alice')
>>>store.append('names', 'Bob')
>>>store.recall('names')
['Alice', 'Bob']
```

bond(*id=None*)

Creates or uses resource required for the permanent back-end

Parameters **id** (*str*) – the unique identifier of the related space

This function should be expanded in sub-class, where necessary.

This function is the right place to create files, databases, and index that can be necessary for a store back-end.

Example:

```
def bond(self, id=None):
    db.execute("CREATE TABLE ...
```

check()

Checks configuration

This function should be expanded in sub-class, where necessary.

This function is the right place to check parameters that can be used by this instance.

Example:

```
def check(self):
    self.context.check(self.prefix+'.db', 'store.db')
```

decrement(*key*, *delta=1*)

Decrements a value

Parameters

- **key** (*str*) – name of the value
- **delta** (*int*) – decrement to apply

Returns the new value

Example:

```
value = store.decrement('gauge')
```

forget(*key=None*)

Forgets a value or all values

Parameters **key** (*str*) – name of the value to forget, or None

To clear only one value, provides the name of it. For example:

```
store.forget('parameter_123')
```

To clear all values in the store, just call the function without a value. For example:

```
store.forget()
```

This function is safe on multiprocessing and multithreading.

from_text(*textual*)

Retrieves a value from a textual representation

Parameters **textual** (*str*) – a textual representation that can be saved in store

Returns a python object

Return type object or None

Here we use `json.loads()` to do the job. You can override this function in your subclass if needed.

increment(*key*, *delta=1*)

Increments a value

Parameters

- **key** (*str*) – name of the value
- **delta** (*int*) – increment to apply

Returns the new value

Example:

```
value = store.increment('gauge')
```

on_init(***kwargs*)

Adds processing to initialization

This function should be expanded in sub-class, where necessary.

This function is the right place to capture additional parameters provided on instance initialisation.

Example:

```
def on_init(self, prefix='sqlite', **kwargs):
    ...
```

recall(*key*, *default=None*)

Recalls a value

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
value = store.recall('parameter_123')
```

This function is safe on multiprocessing and multithreading.

remember(*key*, *value*)

Remembers a value

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – actual value

This functions stores or updates a value in the back-end storage system.

Example:

```
store.remember('parameter_123', 'George')
```

This function is safe on multiprocessing and multithreading.

to_text(*value*)

Turns a value to a textual representation

Parameters **value** (*object*) – a python object that can be serialized

Returns a textual representation that can be saved in store

Return type str

Here we use `json.dumps()` to do the job. You can override this function in your subclass if needed.

update(*key*, *label*, *item*)

Updates a dict

Parameters

- **key** (*str*) – name of the dict
- **label** (*str*) – named entry in the dict
- **item** (*any serializable type is accepted*) – new value of this entry

Example:

```
>>>store.update('input', 'PO Number', '1234A')
>>>store.recall('input')
{'PO Number': '1234A'}
```

shellbot.stores.memory module

class shellbot.stores.memory.**MemoryStore**(*context=None*, ***kwargs*)

Bases: *shellbot.stores.base.Store*

Stores data for one space

This is a key-value store, that supports concurrency across multiple processes.

Example:

```
store = MemoryStore()
```

on_init(***kwargs*)

Adds processing to initialization

shellbot.stores.sqlite module

class shellbot.stores.sqlite.**SqliteStore**(*context=None*, ***kwargs*)

Bases: *shellbot.stores.base.Store*

Stores data for one space

This is a basic permanent key-value store.

Example:

```
store = SqliteStore(db='shellstore.db', id=space.id)
```

bond(*id=None*)

Creates or uses a file to store data

Parameters **id** (*str*) – the unique identifier of the related space

check()

Checks configuration

get_db()
Gets a handle on the database

on_init(*prefix='sqlite'*, *id=None*, *db=None*, ***kwargs*)
Adds processing to initialization

Parameters

- **prefix** (*str*) – the main keyword for configuration of this space
- **id** (*str*) – the unique identifier of the related space (optional)
- **db** (*str*) – name of the file that contains Sqlite data (optional)

Example:

```
store = SqliteStore(context=context, prefix='sqlite')
```

Here we create a new store powered by Sqlite, and use settings under the key `sqlite` in the context of this bot.

Module contents

class shellbot.stores.**Store**(*context=None*, ***kwargs*)
Bases: object

Stores data for one space

This is a key-value store, that supports concurrency across multiple processes.

Configuration of the storage engine is coming from settings of the overall bot.

Example:

```
store = Store(context=my_context)
```

Normally a store is related to one single space. For this, you can use the function `bond()` to set the space unique id.

Example:

```
store.bond(id=space.id)
```

Once this is done, the store can be used to remember and to recall values.

Example:

```
store.remember('gauge', gauge)
...
gauge = store.recall('gauge')
```

append(*key*, *item*)
Appends an item to a list

Parameters

- **key** (*str*) – name of the list
- **item** (*any serializable type is accepted*) – a new item to append

Example:

```
>>>store.append('names', 'Alice')
>>>store.append('names', 'Bob')
>>>store.recall('names')
['Alice', 'Bob']
```

bond(*id=None*)

Creates or uses resource required for the permanent back-end

Parameters **id** (*str*) – the unique identifier of the related space

This function should be expanded in sub-class, where necessary.

This function is the right place to create files, databases, and index that can be necessary for a store back-end.

Example:

```
def bond(self, id=None):
    db.execute("CREATE TABLE ...
```

check()

Checks configuration

This function should be expanded in sub-class, where necessary.

This function is the right place to check parameters that can be used by this instance.

Example:

```
def check(self):
    self.context.check(self.prefix+'.db', 'store.db')
```

decrement(*key*, *delta=1*)

Decrements a value

Parameters

- **key** (*str*) – name of the value
- **delta** (*int*) – decrement to apply

Returns the new value

Example:

```
value = store.decrement('gauge')
```

forget(*key=None*)

Forgets a value or all values

Parameters **key** (*str*) – name of the value to forget, or None

To clear only one value, provides the name of it. For example:

```
store.forget('parameter_123')
```

To clear all values in the store, just call the function without a value. For example:

```
store.forget()
```

This function is safe on multiprocessing and multithreading.

from_text (*textual*)
Retrieves a value from a textual representation

Parameters **textual** (*str*) – a textual representation that can be saved in store

Returns a python object

Return type object or None

Here we use `json.loads()` to do the job. You can override this function in your subclass if needed.

increment (*key*, *delta=1*)
Increments a value

Parameters

- **key** (*str*) – name of the value
- **delta** (*int*) – increment to apply

Returns the new value

Example:

```
value = store.increment('gauge')
```

on_init (**kwargs*)
Adds processing to initialization

This function should be expanded in sub-class, where necessary.

This function is the right place to capture additional parameters provided on instance initialisation.

Example:

```
def on_init(self, prefix='sqlite', **kwargs):
    ...
```

recall (*key*, *default=None*)
Recalls a value

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
value = store.recall('parameter_123')
```

This function is safe on multiprocessing and multithreading.

remember (*key*, *value*)
Remembers a value

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – actual value

This functions stores or updates a value in the back-end storage system.

Example:

```
store.remember('parameter_123', 'George')
```

This function is safe on multiprocessing and multithreading.

to_text(*value*)
Turns a value to a textual representation

Parameters **value** (*object*) – a python object that can be serialized

Returns a textual representation that can be saved in store

Return type str

Here we use `json.dumps()` to do the job. You can override this function in your subclass if needed.

update(*key*, *label*, *item*)
Updates a dict

Parameters

- **key** (*str*) – name of the dict
- **label** (*str*) – named entry in the dict
- **item** (*any serializable type is accepted*) – new value of this entry

Example:

```
>>>store.update('input', 'PO Number', '1234A')
>>>store.recall('input')
{'PO Number': '1234A'}
```

class shellbot.stores.**MemoryStore**(*context=None*, ***kwargs*)
Bases: *shellbot.stores.base.Store*

Stores data for one space

This is a key-value store, that supports concurrency across multiple processes.

Example:

```
store = MemoryStore()
```

on_init(***kwargs*)
Adds processing to initialization

class shellbot.stores.**SqliteStore**(*context=None*, ***kwargs*)
Bases: *shellbot.stores.base.Store*

Stores data for one space

This is a basic permanent key-value store.

Example:

```
store = SqliteStore(db='shellstore.db', id=space.id)
```

bond(*id=None*)
Creates or uses a file to store data

Parameters **id** (*str*) – the unique identifier of the related space

check()
Checks configuration

get_db()
Gets a handle on the database

on_init(*prefix='sqlite'*, *id=None*, *db=None*, ***kwargs*)
Adds processing to initialization

Parameters

- **prefix** (*str*) – the main keyword for configuration of this space
- **id** (*str*) – the unique identifier of the related space (optional)
- **db** (*str*) – name of the file that contains Sqlite data (optional)

Example:

```
store = SqliteStore(context=context, prefix='sqlite')
```

Here we create a new store powered by Sqlite, and use settings under the key `sqlite` in the context of this bot.

shellbot.updaters package

Submodules

shellbot.updaters.base module

class shellbot.updaters.base.**Updater**(*engine=None*, ***kwargs*)
Bases: object

Handles inbound events

Updaters are useful for logging or replication, or side storage, or achiving, of received events.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = FileUpdater(path='/var/log/shellbot.log')
listener = Listener(filter=updater.filter)
```

Here events are written down to a flat file, yet multiple updaters are available.

For example, push every event to Elasticsearch:

```
updater = ElasticsearchUpdater()
listener = Listener(filter=updater.filter)
```

There is also an updater where events are written to a separate Cisco Spark room. This will be useful in cases where safety or control are specifically important.

We are looking for new updaters, so please have a careful look at this file and consider to submit your own module.

filter(*event*)
Filters events handled by listener

Parameters **event** (*Event or Message or Join or Leave, etc.*) – an event received by listener

Returns a filtered event

This function implements the actual auditing of incoming events.

format(*event*)
Prepares an outbound line

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

Returns outbound line

Return type str

This function adapts inbound events to the appropriate format. It turns an object with multiple attributes to a single string that can be saved in a log file.

on_bond(*bot*)
Reacts on space bonding

This function should be expanded in sub-class, where necessary.

Example:

```
def on_bond(self, bot):
    self.db = Driver.open(bot.id)
```

on_dispose()
Reacts on space disposal

This function should be expanded in sub-class, where necessary.

Example:

```
def on_disposal(self):
    self.db = Driver.close()
```

on_init(***kwargs*)
Handles extended initialisation parameters

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='secondary.space', **kwargs):
    ...
```

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

The default behaviour is to write text to `sys.stdout` so it is easy to redirect the stream for any reason.

shellbot.updaters.elastic module

class shellbot.updaters.elastic.**ElasticsearchUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to Elasticsearch

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = ElasticsearchUpdater(host='db.local:9200')
listener = Listener(filter=updater.filter)
```

get_host()
Provides the Elasticsearch host

Return type str

on_bond(*bot*)
Creates index on space bonding

on_init(*host=None*, *index=None*, ***kwargs*)
Writes inbound events to Elasticsearch

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

The function writes the event as a JSON document in Elasticsearch.

shellbot.updaters.file module

class shellbot.updaters.file.**FileUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to a file

This updater serializes events and write JSON records to a flat file.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = FileUpdater(path='/var/log/my_app.log')
listener = Listener(filter=updater.filter)
```

get_path()
Provides the path to the target file

Return type str

on_bond(*bot*)
Creates path on space bonding

on_init(*path=None*, ***kwargs*)
Writes inbound events to a file

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

The function serializes the event and write it to a file.

shellbot.updaters.queue module

class shellbot.updaters.queue.**QueueUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to a queue

This updater serializes events and write them to a queue.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = QueueUpdater(queue=Queue())
listener = Listener(filter=updater.filter)
```

Of course, some process has to grab content from *updater.queue* afterwards.

on_init(*queue=None*, ***kwargs*)
Writes inbound events to a queue

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

This function serializes the event and write it to a queue.

shellbot.updaters.space module

class shellbot.updaters.space.**SpaceUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Replicates messages to a secondary space

format(*event*)
Prepares an outbound line

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

Returns outbound line

Return type str

This function adapts inbound events to the appropriate format. It turns an object with multiple attributes to a single string that can be pushed to a Cisco Spark room.

on_init(*space=None*, *speaker=None*, ***kwargs*)
Replicates messages to a secondary space

Parameters

- **space** (*Space*) – the target space to use (optional)
- **speaker** (*Speaker*) – the speaker instance to use (optional)

Parameters are provided mainly for test injection.

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

With this class a string representation of the received event is forwarded to the speaker queue of a chat space.

Module contents

class shellbot.updaters.**ElasticsearchUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to Elasticsearch

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = ElasticsearchUpdater(host='db.local:9200')
listener = Listener(filter=updater.filter)
```

get_host()
Provides the Elasticsearch host

Return type str

on_bond(*bot*)
Creates index on space bonding

on_init(*host=None*, *index=None*, ***kwargs*)
Writes inbound events to Elasticsearch

put(*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

The function writes the event as a JSON document in Elasticsearch.

class shellbot.updaters.**FileUpdater**(*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to a file

This updater serializes events and write JSON records to a flat file.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = FileUpdater(path='/var/log/my_app.log')
listener = Listener(filter=updater.filter)
```

get_path()
Provides the path to the target file

Return type str

on_bond(*bot*)
Creates path on space bonding

on_init(*path=None*, ***kwargs*)
Writes inbound events to a file

put (*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

The function serializes the event and write it to a file.

class shellbot.updaters.**QueueUpdater** (*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Writes inbound events to a queue

This updater serializes events and write them to a queue.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = QueueUpdater(queue=Queue())
listener = Listener(filter=updater.filter)
```

Of course, some process has to grab content from *updater.queue* afterwards.

on_init (*queue=None*, ***kwargs*)
Writes inbound events to a queue

put (*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – inbound event

This function serializes the event and write it to a queue.

class shellbot.updaters.**SpaceUpdater** (*engine=None*, ***kwargs*)
Bases: *shellbot.updaters.base.Updater*

Replicates messages to a secondary space

format (*event*)
Prepares an outbound line

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

Returns outbound line

Return type str

This function adapts inbound events to the appropriate format. It turns an object with multiple attributes to a single string that can be pushed to a Cisco Spark room.

on_init (*space=None*, *speaker=None*, ***kwargs*)
Replicates messages to a secondary space

Parameters

- **space** (*Space*) – the target space to use (optional)
- **speaker** (*Speaker*) – the speaker instance to use (optional)

Parameters are provided mainly for test injection.

put (*event*)
Processes one event

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

With this class a string representation of the received event is forwarded to the speaker queue of a chat space.

class shellbot.updaters.**Updater**(*engine=None*, ***kwargs*)

Bases: object

Handles inbound events

Updaters are useful for logging or replication, or side storage, or achiving, of received events.

An event may be a Message, a Join or Leave notification, or any other Event.

Updaters expose a filtering function that can be connected to the inbound flow of events handled by the Listener.

Example:

```
updater = FileUpdater(path='/var/log/shellbot.log')
listener = Listener(filter=updater.filter)
```

Here events are written down to a flat file, yet multiple updaters are available.

For example, push every event to Elasticsearch:

```
updater = ElasticsearchUpdater()
listener = Listener(filter=updater.filter)
```

There is also an updater where events are written to a separate Cisco Spark room. This will be useful in cases where safety or control are specifically important.

We are looking for new updaters, so please have a careful look at this file and consider to submit your own module.

filter(*event*)

Filters events handled by listener

Parameters **event** (*Event or Message or Join or Leave, etc.*) – an event received by listener

Returns a filtered event

This function implements the actual auditing of incoming events.

format(*event*)

Prepares an outbound line

Parameters **event** (*Event or Message or Join or Leave*) – an inbound event

Returns outbound line

Return type str

This function adapts inbound events to the appropriate format. It turns an object with multiple attributes to a single string that can be saved in a log file.

on_bond(*bot*)

Reacts on space bonding

This function should be expanded in sub-class, where necessary.

Example:

```
def on_bond(self, bot):
    self.db = Driver.open(bot.id)
```

on_dispose()
Reacts on space disposal

This function should be expanded in sub-class, where necessary.

Example:

```
def on_disposal(self):
    self.db = Driver.close()
```

on_init(***kwargs*)
Handles extended initialisation parameters

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self, prefix='secondary.space', **kwargs):
    ...
```

put(*event*)
Processes one event

Parameters **event** (Event *or* Message *or* Join *or* Leave) – inbound event

The default behaviour is to write text to `sys.stdout` so it is easy to redirect the stream for any reason.

Submodules

shellbot.bot module

class shellbot.bot.**ShellBot**(*engine*, *channel_id=None*, *space=None*, *store=None*, *fan=None*, *machine=None*)
Bases: object

Manages interactions with one space, one store, one state machine

A bot consists of multiple components devoted to one chat channel: - a space - a store - a state machine - ... other optional components that may prove useful

It is designated by a unique id, that is also the unique id of the channel itself.

A bot relies on an underlying engine instance for actual access to the infrastructure, including configuration settings.

The life cycle of a bot can be described as follows:

1.A bot is commonly created from the engine, or directly:

```
bot = ShellBot(engine, channel_id='123')
```

2.The space is connected to some back-end API:

```
space.connect()
```

3.Multiple channels can be handled by a single space:

```
channel = space.create(title)

channel = space.get_by_title(title)
channel = space.get_by_id(id)

channel.title = 'A new title'
space.update(channel)

space.delete(id)
```

Channels feature common attributes, yet can be extended to convey specificities of some platforms.

4.Messages can be posted:

```
space.post_message(id, 'Hello, World!')
```

5.The interface allows for the addition or removal of channel participants:

```
space.add_participants(id, persons)
space.add_participant(id, person, is_moderator)
space.remove_participants(id, persons)
space.remove_participant(id, person)
```

add_participant(*person*, *is_moderator=False*)

Adds one participant

Parameters **person** (*str*) – e-mail addresses of person to add

The underlying platform may, or not, take the optional parameter `is_moderator` into account. The default bahaviour is to discard it, as if the parameter had the value `False`.

add_participants(*persons=[]*)

Adds multiple participants

Parameters **persons** (*list of str*) – e-mail addresses of persons to add

append(*key*, *item*)

Appends an item to a list

Parameters

- **key** (*str*) – name of the list
- **item** (*any serializable type is accepted*) – a new item to append

Example:

```
>>>bot.append('names', 'Alice')
>>>bot.append('names', 'Bob')
>>>bot.recall('names')
['Alice', 'Bob']
```

bond()

Bonds to a channel

This function is called either after the creation of a new channel, or when the bot has been invited to an existing channel. In such situations the banner should be displayed as well.

There are also situations where the engine has been completely restarted. The bot bonds to a channel where it has been before. In that case the banner should be avoided.

dispose(***kwargs*)
Disposes all resources

This function deletes the underlying channel in the cloud and resets this instance. It is useful to restart a clean environment.

>>>bot.bond(title="Working Space") ... >>>bot.dispose()

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

forget(*key=None*)
Forgets a value or all values

Parameters **key** (*str*) – name of the value to forget, or None

To clear only one value, provides the name of it. For example:

```
bot.forget('variable_123')
```

To clear all values in the store, just call the function without a value. For example:

```
bot.forget()
```

id
Gets unique id of the related chat channel

Returns the id of the underlying channel, or None

is_ready
Checks if this bot is ready for interactions

Returns True or False

on_bond()
Adds processing to channel bonding

This function should be changed in sub-class, where necessary.

Example:

```
def on_bond(self):
    do_something_important_on_bond()
```

on_enter()
Enters a channel

on_exit()
Exits a channel

on_init()
Adds to bot initialization

It can be overlaid in subclass, where needed

on_reset()
Adds processing to space reset

This function should be expanded in sub-class, where necessary.

Example:

```
def on_reset(self):
    self._last_message_id = 0
```

recall (*key*, *default=None*)
Recalls a value

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
value = bot.recall('variable_123')
```

remember (*key*, *value*)
Remembers a value

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

This functions stores or updates a value in the back-end storage system.

Example:

```
bot.remember('variable_123', 'George')
```

remove_participant (*person*)
Removes one participant

Parameters **person** (*str*) – e-mail addresses of person to add

remove_participants (*persons=[]*)
Removes multiple participants

Parameters **persons** (*list of str*) – e-mail addresses of persons to remove

reset ()
Resets a space

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

say (*text=None*, *content=None*, *file=None*, *person=None*)
Sends a message to the chat space

Parameters

- **text** (*str or None*) – Plain text message
- **content** (*str or None*) – Rich content such as Markdown or HTML
- **file** (*str or None*) – path or URL to a file to attach
- **person** (*str*) – for direct message to someone

say_banner ()
Sends banner to the channel

This function uses following settings from the context:

- `bot.banner.text` or `bot.on_enter` - a textual message
- `bot.banner.content` - some rich content, e.g., Markdown or HTML

- `bot.banner.file` - a document to be uploaded

The quickest setup is to change `bot.on_enter` in settings, or the environment variable `$BOT_ON_ENTER`.

Example:

```
os.environ['BOT_ON_ENTER'] = 'You can now chat with Batman'
engine.configure()
```

Then there are situtations where you want a lot more flexibility, and rely on a smart banner. For example you could do the following:

```
settings = {
    'bot': {
        'banner': {
            'text': u"Type '@{} help' for more information",
            'content': u"Type ``@{} help`` for more information",
            'file': "http://on.line.doc/guide.pdf"
        }
    }
}

engine.configure(settings)
```

When bonding to a channel, the bot will send an update similar to the following one, with a nice looking message and image:

```
Type '@Shelly help' for more information
```

Default settings for the banner rely on the environment, so it is easy to inject strings from the outside. Use following variables:

- `$BOT_BANNER_TEXT` or `$BOT.ON_ENTER` - the textual message
- `$BOT_BANNER_CONTENT` - some rich content, e.g., Markdown or HTML
- `$BOT_BANNER_FILE` - a document to be uploaded

title

Gets title of the related chat channel

Returns the title of the underlying channel, or None

update (*key*, *label*, *item*)

Updates a dict

Parameters

- **key** (*str*) – name of the dict
- **label** (*str*) – named entry in the dict
- **item** (*any serializable type is accepted*) – new value of this entry

Example:

```
>>>bot.update('input', 'PO Number', '1234A')
>>>bot.update('input', 'description', 'some description')
>>>bot.recall('input')
{'PO Number': '1234A',
 'description': 'some description'}
```

shellbot.bus module

class shellbot.bus.**Bus**(*context*)

Bases: object

Represents an information bus between publishers and subscribers

In the context of shellbot, channels are channel identifiers, and messages are python objects serializable with json.

A first pattern is the synchronization of direct channels from the group channel:

- every direct channel is a subscriber, and filters messages sent to their own channel identifier
- group channel is a publisher, and broadcast instructions to the list of direct channel identifiers it knows about

A second pattern is the observation by a group channel of what is happening in related direct channels:

- every direct channel is a publisher, and the channel used is their own channel identifier
- group channel is a subscriber, and observed messages received from all direct channels it knows about

For example, a distributed voting system can be built by combining the two patterns. The vote itself can be triggered simultaneously to direct channels on due time, so that every participants are involved more or less at the same time. And data that is collected in direct channels can ce centralised back to the group channel where results are communicated.

DEFAULT_ADDRESS = 'tcp://127.0.0.1:5555'

check()

Checks configuration settings

This function reads key `bus` and below, and update the context accordingly. It handles following parameters:

- `bus.address` - focal point of bus exchanges on the network. The default value is `tcp://*:5555` which means 'use TCP port 5555 on local machine'.

publish()

Publishes messages

Returns Publisher

Example:

```
# get a publisher for subsequent broadcasts
publisher = bus.publish()

# start the publishing process
publisher.start()

...

# broadcast information_message
publisher.put(channel, message)
```

subscribe(*channels*)

Subcribes to some channels

Parameters **channels** (*str or list of str*) – one or multiple channels

Returns Subscriber

Example:

```
# subscribe from all direct channels related to this group channel
subscriber = bus.subscribe(bot.direct_channels)

...

# get next message from these channels
message = subscriber.get()
```

class shellbot.bus.**Publisher**(*context*)

Bases: multiprocessing.process.Process

Publishes asynchronous messages

For example, from a group channel, you may send instructions to every direct channels:

```
# get a publisher
publisher = bus.publish()

# send instruction to direct channels
publisher.put(bot.direct_channels, instruction)
```

From within a direct channel, you may reflect your state to observers:

```
# get a publisher
publish = bus.publish()

# share new state
publisher.put(bot.id, bit_of_information_here)
```

DEFER_DURATION = 0.3

EMPTY_DELAY = 0.005

process(*item*)

Processes items received from the queue

Parameters **item** (*str*) – the item received

Note that the item should result from serialization of (channel, message) tuple done previously.

put(*channels*, *message*)

Broadcasts a message

Parameters

- **channels** (*str or list of str*) – one or multiple channels
- **message** (*dict or other json-serializable object*) – the message to send

Example:

```
message = { ... }
publisher.put(bot.id, message)
```

This function actually put the message in a global queue that is handled asynchronously. Therefore, when the function returns there is no guarantee that message has been transmitted nor received.

run()

Continuously broadcasts messages

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
publisher = Publisher(address)
process = publisher.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when a poison pill is pushed to the queue. For example:

```
publisher.fan.put(None)
```

class `shellbot.bus.`**`Subscriber`**(*context*, *channels*)

Bases: `object`

Subscribes to asynchronous messages

For example, from a group channel, you may subscribe from direct channels of all participants:

```
# subscribe from all direct channels related to this group channel
subscriber = bus.subscribe(bot.direct_channels)

# get messages from direct channels
while True:
    message = subscriber.get()
    ...
```

From within a direct channel, you may receive instructions sent by the group channel:

```
# subscribe for messages sent to me
subscriber = bus.subscribe(bot.id)

# get and process instructions one at a time
while True:
    instruction = subscriber.get()
    ...
```

`get`(*block=False*)

Gets next message

Returns dict or other serializable message or None

This function returns next message that has been made available, or None if no message has arrived yet.

Example:

```
message = subscriber.get()  # immedaite return
if message:
    ...
```

Change the parameter `block` if you prefer to wait until next message arrives.

Example:

```
message = subscriber.get(block=True)  # wait until available
```

Note that this function does not preserve the enveloppe of the message. In other terms, the channel used for the communication is lost in translation. Therefore the need to put within messages all information that may be relevant for the receiver.

shellbot.channel module

class shellbot.channel.**Channel**(*attributes=None*)

Bases: object

Represents a chat channel managed by a space

A channel is a group of interactions within a chat space. It features a unique identifier, and a title. It also have participants and content.

This class is a general abstraction of a communication channel, that can easily been adapted to various chat systems. It has been designed as a dictionary wrapper, with minimum exposure to shellbot, while enabling the transmission of rich information through serialization.

Instances of Channel are created within a Space object, and consumed by it as well.

For example, to create a channel with a given title, you could write:

```
channel = space.create(title='A new channel')
bot.say(u"I am happy to join {}".format(channel.title))
```

And to change the title of the channel:

```
channel.title = 'An interesting place'
space.update(channel)
```

Direct channels support one-to-one interactions between the bot and one person. The creation of a direct channel can only be indirect, by sending an invitation to the target person. For example:

```
bot.say(person='foo.bar@acme.com',
        text='Do you want to deal with me?')
```

If the person receives and accepts the invitation, the engine will receive a `join` event and load a new bot devoted to the direct channel. So, at the end of the day, when multiple persons interact with a shellbot, this involve both group and direct channels.

For example, if you create a shellbot named shelly, that interacts with Alice and with Bob, then shelly will overlook multiple bots and channels:

- shelly main channel (bot + channel + store + state machine)
- direct channel with Alice (bot + channel + store + state machine)
- direct channel with Bob (bot + channel + store + state machine)

get(*key*, *default=None*)

Returns the value of one attribute :param key: name of the attribute :type key: str

Parameters **default** (*str or other serializable object*) – default value of the attribute

Returns value of the attribute

Return type str or other serializable object or None

The use case for this function is when you adapt a channel that does not feature an attribute that is expected by shellbot. More specifically, call this function on optional attributes so as to avoid AttributeError

id
Returns channel unique id

Return type str

is_direct
Indicates if this channel is only for one person and the bot

Return type str

A channel is deemed direct when it is reserved to one-to-one interactions. Else it is considered a group channel, with potentially many participants.

is_moderated
Indicates if this channel is moderated

Return type str

A channel is moderated when some participants have specific powers that others do not have. Else all participants are condidered the same and peer with each others.

title
Returns channel title

Return type str

shellbot.context module

class shellbot.context.**Context**(*settings=None*, *filter=None*)
Bases: object

Stores settings across multiple independent processing units

This is a key-value store, that supports concurrency across multiple processes.

apply(*settings={}*)
Applies multiple settings at once

Parameters **settings** (*dict*) – variables to be added to this context

check(*key*, *default=None*, *is_mandatory=False*, *validate=None*, *filter=False*)
Checks some settings

Parameters

- **key** – the key that has to be checked
- **default** (*str*) – the default value if no statement can be found
- **is_mandatory** (*bool*) – raise an exception if keys are not found
- **validate** (*callable*) – a function called to validate values before the import
- **filter** (*bool*) – look at the content, and change it eventually

Example:

```
context = Context({
    'spark': {
        'room': 'My preferred room',
        'participants':
```

```
            ['alan.droit@azerty.org', 'bob.nard@support.tv'],
        'team': 'Anchor team',
        'token': 'hkNWEtMJNkODk3ZDZLOGQ0OVGlZWU1NmYtyY>',
        'webhook': "http://73a1e282.ngrok.io",
        'weird_token', '$WEIRD_TOKEN',
    }
})

context.check('spark.room', is_mandatory=True)
context.check('spark.team')
context.check('spark.weird_token', filter=True)
```

When a default value is provided, it is used to initialize properly a missing key:

```
context.check('general.switch', 'on')
```

Another usage is to ensure that a key has been set:

```
context.check('spark.room', is_mandatory=True)
```

Additional control can be added with the validation function:

```
context.check('general.switch',
              validate=lambda x: x in ('on', 'off'))
```

When filter is True, if the value is a string starting with '$', then a variable with the same name is loaded from the environment:

```
>>>token=context.check('spark.weird_token', filter=True)
>>>assert token == os.environ.get('WEIRD_TOKEN')
True
```

The default filter can be changed at the creation of a context:

```
>>>context=Context(filter=lambda x : x + '...')
```

This function raises `KeyError` if a mandatory key is absent. If a validation function is provided, then a `ValueError` can be raised as well in some situations.

clear()
Clears content of a context

decrement(*key*, *delta=1*)
Decrements a value

get(*key*, *default=None*)
Retrieves the value of one configurationkey

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
message = context.get('bot.on_start')
```

This function is safe on multiprocessing and multithreading.

has (*prefix*)

Checks the presence of some prefix

Parameters **prefix** (*str*) – key prefix to be checked

Returns True if one or more key start with the prefix, else False

This function looks at keys actually used in this context, and return True if prefix is found. Else it returns False.

Example:

```
context = Context(settings={'space': {'title', 'a title'}})

>>>context.has('space')
True

>>>context.has('space.title')
True

>>>context.has('spark')
False
```

increment (*key*, *delta=1*)

Increments a value

is_empty

Does the context store something?

Returns True if there at least one value, False otherwise

set (*key*, *value*)

Changes the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

Example:

```
context.set('bot.on_start', 'hello world')
```

This function is safe on multiprocessing and multithreading.

classmethod set_logger (*level=10*)

Configure logging

Parameters **level** – expected level of verbosity

This utility function should probably be put elsewhere

shellbot.engine module

class shellbot.engine.**Engine**(*context=None*, *settings={}*, *configure=False*, *mouth=None*, *ears=None*, *fan=None*, *space=None*, *type=None*, *server=None*, *store=None*, *command=None*, *commands=None*, *driver=<class 'shellbot.bot.ShellBot'>*, *machine_factory=None*, *updater_factory=None*, *preload=0*)

Bases: object

Powers multiple bots

The engine manages the infrastructure that is used accross multiple bots acting in multiple spaces. It is made of an extensible set of components that share the same context, that is, configuration settings.

Shellbot allows the creation of bots with a given set of commands. Each bot instance is bonded to a single chat space. The chat space can be either created by the bot itself, or the bot can join an existing space.

The first use case is adapted when a collaboration space is created for semi-automated interactions between human and machines. In the example below, the bot controls the entire life cycle of the chat space. A chat space is created when the program is launched. And it is deleted when the program is stopped.

Example of programmatic chat space creation:

```
from shellbot import Engine, ShellBot, Context, Command
Context.set_logger()

# create a bot and load command
#
class Hello(Command):
    keyword = 'hello'
    information_message = u"Hello, World!"

engine = Engine(command=Hello(), type='spark')

# load configuration
#
engine.configure()

# create a chat space, or connect to an existing one
# settings of the chat space are provided
# in the engine configuration itself
#
engine.bond(reset=True)

# run the engine
#
engine.run()

# delete the chat channel when the engine is stopped
#
engine.dispose()
```

A second interesting use case is when a bot is invited to an existing chat space. On such an event, a new bot instance can be created and bonded to the chat space.

Example of invitation to a chat space:

```
def on_enter(self, channel_id):
    bot = engine.get_bot(channel_id=channel_id)
```

The engine is configured by setting values in the context that is attached to it. This is commonly done by loading the context with a dict before the creation of the engine itself, as in the following example:

```
context = Context({

    'bot': {
        'on_enter': 'You can now chat with Batman',
        'on_exit': 'Batman is now quitting the channel, bye',
    },

    'server': {
        'url': 'http://d9b62df9.ngrok.io',
        'hook': '/hook',
    },

})

engine = Engine(context=context)

engine.configure()
```

Please note that the configuration is checked and actually used on the call `engine.configure()`, rather on the initialisation itself.

When configuration statements have been stored in a separate text file in YAML format, then the engine can be initialised with an empty context, and configuration is loaded afterwards.

Example:

```
engine = Engine()
engine.configure_from_path('/opt/shellbot/my_bot.yaml')
```

When no configuration is provided to the engine, then default settings are considered for the engine itself, and for various components.

For example, for a basic engine interacting in a Cisco Spark channel:

```
engine = Engine(type='spark')
engine.configure()
```

When no indication is provided at all, the engine loads a space of type 'local'.

So, in other terms:

```
engine = Engine()
engine.configure()
```

is strictly equivalent to:

```
engine = Engine('local')
engine.configure()
```

In principle, the configuration of the engine is set once for the full life of the instance. This being said, some settings can be changed globally with the member function *set()*. For example:

```
engine.set('bot.on_banner': 'Hello, I am here to help')
```

DEFAULT_SETTINGS = {'bot': {'banner.content': '$BOT_BANNER_CONTENT', 'on_enter': '$BOT_ON_ENTER', 'on

bond(*title=None*, *reset=False*, *participants=None*, ***kwargs*)
Bonds to a channel

Parameters

- **title** – title of the target channel
- **reset** (*bool*) – if True, delete previous channel and re-create one
- **participants** (*list of str*) – the list of initial participants (optional)

Type title: str

Returns Channel or None

This function creates a channel, or connect to an existing one. If no title is provided, then the generic title configured for the underlying space is used instead.

For example:

```
channel = engine.bond('My crazy channel')
if channel:
    ...
```

Note: this function asks the listener to load a new bot in its cache on successful channel creation or lookup. In other terms, this function can be called safely from any process for the creation of a channel.

build_bot(*id=None*, *driver=<class 'shellbot.bot.ShellBot'>*)
Builds a new bot

Parameters **id** (*str*) – The unique id of the target space

Returns a ShellBot instance, or None

This function receives the id of a chat space, and returns the related bot.

build_machine(*bot*)
Builds a state machine for this bot

Parameters **bot** (ShellBot) – The target bot

Returns a Machine instance, or None

This function receives a bot, and returns a state machine bound to it.

build_store(*channel_id=None*)
Builds a store for this bot

Parameters **channel_id** (*str*) – Identifier of the target chat space

Returns a Store instance, or None

This function receives an identifier, and returns a store bound to it.

build_updater(*id*)
Builds an updater for this channel

Parameters **id** (*str*) – The identifier of an audited channel

Returns an Updater instance, or None

This function receives a bot, and returns a state machine bound to it.

check()
Checks settings of the engine

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key `bot` and below, and update the context accordingly.

Example:

```
context = Context({

    'bot': {
        'on_enter': 'You can now chat with Batman',
        'on_exit': 'Batman is now quitting the channel, bye',
    },

    'server': {
        'url': 'http://d9b62df9.ngrok.io',
        'hook': '/hook',
    },

})
engine = Engine(context=context)
engine.check()
```

configure(*settings={}*)
Checks settings

Parameters **settings** (*dict*) – configuration information

If no settings is provided, and the context is empty, then `self.DEFAULT_SETTINGS` and `self.space.DEFAULT_SETTINGS` are used instead.

configure_from_file(*stream*)
Reads configuration information

Parameters **stream** (`file`) – the handle that contains configuration information

The function loads configuration from the file and from the environment. Port number can be set from the command line.

configure_from_path(*path='settings.yaml'*)
Reads configuration information

Parameters **path** (*str*) – path to the configuration file

The function loads configuration from the file and from the environment. Port number can be set from the command line.

dispatch(*event*, ***kwargs*)
Triggers objects that have registered to some event

Parameters **event** (*str*) – label of the event

Example:

```
def on_bond(self):
    self.dispatch('bond', bot=this_bot)
```

For each registered object, the function will look for a related member function and call it. For example for the event 'bond' it will look for the member function 'on_bond', etc.

Dispatch uses weakref so that it affords the unattended deletion of registered objects.

dispose(*title=None*, ***kwargs*)
Destroys a named channel

Parameters **title** – title of the target channel

Type title: str

enumerate_bots()
Enumerates all bots

get(*key*, *default=None*)
Retrieves the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
message = engine.get('bot.on_start')
```

This function is safe on multiprocessing and multithreading.

get_bot(*channel_id=None*, ***kwargs*)
Gets a bot by id

Parameters **channel_id** (*str*) – The unique id of the target chat space

Returns a bot instance, or None

This function receives the id of a chat space, and returns the related bot.

If no id is provided, then the underlying space is asked to provide with a default channel, as set in overall configuration.

Note: this function should not be called from multiple processes, because this would create one bot per process. Use the function `engine.bond()` for the creation of a new channel.

get_hook()
Provides the hooking function to receive messages from Cisco Spark

hook(*server=None*)
Connects this engine with back-end API

Parameters **server** (Server) – web server to be used

This function adds a route to the provided server, and asks the back-end service to send messages there.

initialize_store(*bot*)
Copies engine settings to the bot store

load_command(**args*, ***kwargs*)
Loads one commands for this bot

This function is a convenient proxy for the underlying shell.

load_commands(**args*, ***kwargs*)
Loads commands for this bot

This function is a convenient proxy for the underlying shell.

name
Retrieves the dynamic name of this bot

Returns The value of `bot.name` key in current context

Return type str

on_build(*bot*)
Extends the building of a new bot instance

Parameters **bot** (ShellBot) – a new bot instance

Provide your own implementation in a sub-class where required.

Example:

```
on_build(self, bot):
    bot.secondary_machine = Input(...)
```

on_enter(*join*)
Bot has been invited to a chat space

Parameters **join** (Join) – The join event received from the chat space

Provide your own implementation in a sub-class where required.

Example:

```
on_enter(self, join):
    mailer.post(u"Invited to {}".format(join.space_title))
```

on_exit(*leave*)
Bot has been kicked off from a chat space

Parameters **leave** (Leave) – The leave event received from the chat space

Provide your own implementation in a sub-class where required.

Example:

```
on_exit(self, leave):
    mailer.post(u"Kicked off from {}".format(leave.space_title))
```

on_start()
Does additional stuff when the engine is started

Provide your own implementation in a sub-class where required.

on_stop()
Does additional stuff when the engine is stopped

Provide your own implementation in a sub-class where required.

Note that some processes may have been killed at the moment of this function call. This is likely to happen when end-user hits Ctl-C on the keyboard for example.

register(*event*, *instance*)
Registers an object to process an event

Parameters

- **event** (*str*) – label, such as 'start' or 'bond'
- **instance** (*object*) – an object that will handle the event

This function is used to propagate events to any module that may need it via callbacks.

On each event, the engine will look for a related member function in the target instance and call it. For example for the event 'start' it will look for the member function 'on_start', etc.

Following standard events can be registered:

- 'bond' - when the bot has connected to a chat channel

- 'dispose' - when resources, including chat space, will be destroyed
- 'start' - when the engine is started
- 'stop' - when the engine is stopped
- 'join' - when a person is joining a space
- 'leave' - when a person is leaving a space

Example:

```
def on_init(self):
    self.engine.register('bond', self)  # call self.on_bond()
    self.engine.register('dispose', self) # call self.on_dispose()
```

If the function is called with an unknown label, then a new list of registered callbacks will be created for this event. Therefore the engine can be used for the dispatching of any custom event.

Example:

```
self.engine.register('input', processor)  # for processor.on_input()
...
received = 'a line of text'
self.engine.dispatch('input', received)
```

Registration uses weakref so that it affords the unattended deletion of registered objects.

run(*server=None*)

Runs the engine

Parameters **server** (Server) – a web server

If a server is provided, it is ran in the background. A server could also have been provided during initialisation, or loaded during configuration check.

If no server instance is available, a loop is started to fetch messages in the background.

In both cases, this function does not return, except on interrupt.

set(*key*, *value*)

Changes the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

Example:

```
engine.set('bot.on_start', 'hello world')
```

This function is safe on multiprocessing and multithreading.

start()

Starts the engine

start_processes()

Starts the engine processes

This function starts a separate process for each main component of the architecture: listener, speaker, etc.

stop()
Stops the engine

This function changes in the context a specific key that is monitored by bot components.

version
Retrieves the version of this bot

Returns The value of `bot.version` key in current context

Return type str

shellbot.events module

class shellbot.events.**Event**(*attributes=None*)
Bases: object

Represents an event received from the chat system

Events, and derivated objects such as instances of Message, abstract pieces of information received from various chat systems. They are designed as dictionary wrappers, with minimum exposure to shellbot, while enabling the transmission of rich information through serialization.

The life cycle of an event starts within a Space instance, most often, in the webhook triggered by a remote chat system. In order to adapt to shellbot, code should build the appropriate event instance, and push it to the queue used by the listener.

Example:

```
item = self.api.messages.get(messageId=message_id)
my_engine.ears.put(Message(item._json))
```

get(*key*, *default=None*)
Returns the value of one attribute :param key: name of the attribute :type key: str

Parameters **default** (*str or other serializable object*) – default value of the attribute

Returns value of the attribute

Return type str or other serializable object or None

The use case for this function is when you adapt an event that does not feature an attribute that is expected by shellbot. More specifically, call this function on optional attributes so as to avoid AttributeError

For example, some Cisco Spark messages may have `toPersonId`, but not all. So you could do:

```
message = Message(received_item)
to_id = message.get('toPersonId')
if to_id:
    ...
```

type = 'event'

class shellbot.events.**EventFactory**
Bases: object

Generates events

classmethod **build_event**(*attributes*)
Turns a dictionary to a typed event

Parameters **attributes** (*dict*) – the set of attributes to consider

Returns an Event, such as a Message, a Join, a Leave, etc.

class shellbot.events.**Join**(*attributes=None*)
Bases: *shellbot.events.Event*

Represents the addition of someone to a space

actor_address
Returns the address of the joining actor

Return type str or None

This attribute can be passed to `add_participant()` if needed.

actor_id
Returns the id of the joining actor

Return type str or None

This attribute allows listener to identify who joins a space.

actor_label
Returns the name or title of the joining actor

Return type str or None

This attribute allows listener to identify who joins a space.

channel_id
Returns the id of the joined space

Return type str or None

stamp
Returns the date and time of this event in ISO format

Return type str or None

type = 'join'

class shellbot.events.**Leave**(*attributes=None*)
Bases: *shellbot.events.Event*

Represents the removal of someone to a space

actor_address
Returns the address of the leaving actor

Return type str or None

This attribute can be passed to `add_participant()` if needed.

actor_id
Returns the id of the leaving actor

Return type str or None

This attribute allows listener to identify who leaves a space.

actor_label
Returns the name or title of the leaving actor

Return type str or None

This attribute allows listener to identify who leaves a space.

channel_id
Returns the id of the left space

Return type str or None

stamp
Returns the date and time of this event in ISO format

Return type str or None

type = 'leave'

class shellbot.events.**Message**(*attributes=None*)
Bases: *shellbot.events.Event*

Represents a message received from the chat system

attachment
Returns name of uploaded file

Return type str

This attribute is set on file upload. It provides with the external name of the file that has been shared, if any.

For example, to get a local copy of an uploaded file:

```
if message.attachment:
    path = space.download_attachment(message.url)
```

channel_id
Returns the id of the chat space

Return type str or None

content
Returns message rich content

Return type str

This function preserves rich content that was used to create the message, be it Markdown, HTML, or something else.

If no rich content is provided, than this attribute is equivalent to `self.text`

from_id
Returns the id of the message originator

Return type str or None

This attribute allows listener to distinguish between messages from the bot and messages from other chat participants.

from_label
Returns the name or title of the message originator

Return type str or None

This attribute is used by updaters that log messages or copy them for archiving.

is_direct
Determines if this is a direct message

Return type True or False

This attribute is set for 1-to-1 channels. It allows the listener to determine if the input is explicitly for this bot or not.

mentioned_ids
Returns the list of mentioned persons

Return type list of str, or []

This attribute allows the listener to determine if the input is explicitly for this bot or not.

stamp
Returns the date and time of this event in ISO format

Return type str or None

This attribute allows listener to limit the horizon of messages fetched from a space back-end.

text
Returns message textual content

Return type str

This function returns a bare string that can be handled directly by the shell. This has no tags nor specific binary format.

type = 'message'

url
Returns link to uploaded file

Return type str

This attribute is set on file upload. It provides with the address that can be used to fetch the actual content.

There is a need to rely on the underlying space to authenticate and get the file itself. For example:

```
if message.url:
    content = space.get_attachment(message.url)
```

shellbot.listener module

class shellbot.listener.**Listener**(*engine=None*, *filter=None*)
Bases: multiprocessing.process.Process

Handles messages received from chat space

DEFER_DURATION = 2.0

EMPTY_DELAY = 0.005

FRESH_DURATION = 0.5

idle()
Finds something smart to do

on_inbound(*received*)
Another event has been received

Parameters **received** (Event *or derivative*) – the event received

Received information is transmitted to registered callbacks on the inbound at the engine level.

on_join(*received*)
A person, or the bot, has joined a space

Parameters **received** (Join) – the event received

Received information is transmitted to registered callbacks on the `join` at the engine level.

In the special case where the bot itself is joining a channel by invitation, then the event `enter` is dispatched instead.

on_leave(*received*)

A person, or the bot, has left a space

Parameters **received** (Leave) – the event received

Received information is transmitted to registered callbacks on the `leave` at the engine level.

In the special case where the bot itself has been kicked off from a channel, then the event `exit` is dispatched instead.

on_message(*received*)

A message has been received

Parameters **received** (Message) – the message received

Received information is transmitted to registered callbacks on the `message` event at the engine level.

When a message is directed to the bot it is submitted directly to the shell. This is handled as a command, that can be executed immediately, or pushed to the inbox and processed by the worker when possible.

All other input is thrown away, except if there is some downwards listeners. In that situation the input is pushed to a queue so that some process can pick it up and process it.

The protocol for downwards listeners works like this:

- Check the `bot.fan` queue frequently
- On each check, update the string `fan.<channel_id>` in the context with the value of `time.time()`. This will say that you are around.

The value of `fan.<channel_id>` is checked on every message that is not for the bot itself. If this is fresh enough, then data is put to the `bot.fan` queue. Else message is just thrown away.

process(*item*)

Processes items received from the chat space

Parameters **item** (*dict or json-encoded string*) – the item received

This function dispatches items based on their type. The type is a key of the provided dict.

Following types are handled:

- `message` – This is a textual message, maybe with a file attached. The message is given to the `on_message()` function.
- `join` – This is when a person or the bot joins a space. The function `on_join()` is called, providing details on the person or the bot who joined
- `leave` – This is when a person or the bot leaves a space. The function `on_leave()` is called with details on the leaving person or bot.
- `load_bot` – This is a special event to load the cache in the process that is running the listener. The identifier of the channel to load is provided as well.
- on any other case, the function `on_inbound()` is called.

run()

Continuously receives updates

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
listener = Listener(engine=my_engine)
process = listener.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when a poison pill is pushed to the queue. For example:

```
engine.ears.put(None)
```

shellbot.observer module

class `shellbot.observer.`**Observer**(*engine=None*)

Bases: `multiprocessing.process.Process`

Dispatches inbound records to downwards updaters

EMPTY_DELAY = 0.005

process(*item*)

Handles one record or command

Parameters **item** (*str or object*) – the record or command

run()

Continuously handle inbound records and commands

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
observer = Observer(engine=my_engine)
observer.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when an exception is pushed to the queue. For example:

```
engine.fan.put(None)
```

shellbot.server module

class `shellbot.server.`**Server**(*context=None*, *httpd=None*, *route=None*, *routes=None*, *check=False*)

Bases: `bottle.Bottle`

Serves web requests

add_route(*item*)
Adds one web route

Parameters **route** (Route) – one additional route

add_routes(*items*)
Adds web routes

Parameters **routes** (*list of routes*) – a list of additional routes

configure(*settings={}*)
Checks settings of the server

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key `server` and below, and update the context accordingly:

```
>>>server.configure({'server': {
       'binding': '10.4.2.5',
       'port': 5000,
       'debug': True,
       }})
```

This can also be written in a more compact form:

```
>>>server.configure({'server.port': 5000})
```

route(*route*)
Gets one route by path

Returns the related route, or None

routes
Lists all routes

Returns a list of routes, or []

Example:

```
>>>server.get_routes()
['/hello', '/world']
```

run()
Serves requests

shellbot.shell module

class shellbot.shell.**Shell**(*engine*)
Bases: object

Parses input and reacts accordingly

command(*keyword*)
Get one command

Parameters **keyword** (*str*) – the keyword for this command

Returns the instance for this command

Return type *command* or None

Lists available commands and related usage information.

Example:

```
>>>print(shell.command('help').information_message)
```

commands
Lists available commands

Returns a list of verbs

Return type list of str

This function provides with a dynamic inventory of all capabilities of this shell.

Example:

```
>>>print(shell.commands)
['*default', '*empty', 'help']
```

configure(*settings={}*)
Checks settings of the shell

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key `shell` and below, and update the context accordingly:

```
>>>shell.configure({'shell': {
       'commands':
          ['examples.exception.state', 'examples.exception.next']
       }})
```

This can also be written in a more compact form:

```
>>>shell.configure({'shell.commands':
       ['examples.exception.state', 'examples.exception.next']
       })
```

Note that this function does preserve commands that could have been loaded previously.

do(*line*, *received=None*)
Handles one line of text

Parameters

- **line** (*str*) – a line of text to parse and to handle
- **received** (Message) – the message that contains the command

This function uses the first token as a verb, and looks for a command of the same name in the shell.

If the command does not exist, the command `*default` is used instead. Default behavior is implemented in `shellbot.commands.default` yet you can load a different command for customization.

If an empty line is provided, the command `*empty` is triggered. Default implementation is provided in `shellbot.commands.empty`.

When a file has been uploaded, the information is given to the command that is executed. If no message is provided with the file, the command `*upload` is triggered instad of `*empty`. Default implementation is provided in `shellbot.commands.upload`.

Following parameters are used for the execution of a command:

- `bot` - A bot instance is retrieved from the channel id mentioned in `received`, and provided to the command.
- `arguments` - This is a string that contains everything after the command verb. When `hello How are you doing?` is submitted to the shell, `hello` is the verb, and `How are you doing?` are the arguments. This is the regular case. If there is no command `hello` then the command `*default` is used instead, and arguments provided are the full line `hello How are you doing?`.
- `attachment` - When a file has been uploaded, this attribute provides its external name, e.g., `picture024.png`. This can be used in the executed command, if you keep in mind that the same name can be used multiple times in a conversation.
- `url` - When a file has been uploaded, this is the handle by which actual content can be retrieved. Usually, ask the underlying space to get a local copy of the document.

load_command(*command*)

Loads one command for this shell

Parameters **command** (*str or command*) – A command to load

If a string is provided, it should reference a python module that can be used as a command. Check `base.py` in `shellbot.commands` for a clear view of what it means to be a vaid command for this shell.

Example:

```
>>>shell.load_command('shellbot.commands.help')
```

If an object is provided, it should duck type the command defined in `base.py` in `shellbot.commands`.

Example:

```
>>>from shellbot.commands.version import Version
>>>command = Version()
>>>shell.load_command(command)
```

load_commands(*commands=[]*)

Loads commands for this shell

Parameters **commands** (*List of labels or list of commands*) – A list of commands to load

Example:

```
>>>commands = ['shellbot.commands.help']
>>>shell.load_commands(commands)
```

Each label should reference a python module that can be used as a command. Check `base.py` in `shellbot.commands` for a clear view of what it means to be a vaid command for this shell.

If objects are provided, they should duck type the command defined in `base.py` in `shellbot.commands`.

Example:

```
>>>from shellbot.commands.version import Version
>>>version = Version()
>>>from shellbot.commands.help import Help
>>>help = Help()
>>>shell.load_commands([version, help])
```

```
```

load_default_commands()
Loads default commands for this shell

Example:

```
>>>shell.load_default_commands()
```

shellbot.speaker module

class shellbot.speaker.**Speaker**(*engine=None*)
Bases: multiprocessing.process.Process

Sends updates to a business messaging space

EMPTY_DELAY = 0.005

process(*item*)
Sends one update to a business messaging space

Parameters **item** (*str or object*) – the update to be transmitted

run()
Continuously send updates

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
speaker = Speaker(engine=my_engine)
speaker.start()
```

The recommended way for stopping the process is to change the parameter general.switch in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when an exception is pushed to the queue. For example:

```
engine.mouth.put(None)
```

class shellbot.speaker.**Vibes**(*text=None*, *content=None*, *file=None*, *channel_id=None*, *person=None*)
Bases: object

Module contents

class shellbot.**Bus**(*context*)
Bases: object

Represents an information bus between publishers and subscribers

In the context of shellbot, channels are channel identifiers, and messages are python objects serializable with json.

A first pattern is the synchronization of direct channels from the group channel:

- every direct channel is a subscriber, and filters messages sent to their own channel identifier
- group channel is a publisher, and broadcast instructions to the list of direct channel identifiers it knows about

A second pattern is the observation by a group channel of what is happening in related direct channels:

- every direct channel is a publisher, and the channel used is their own channel identifier
- group channel is a subscriber, and observed messages received from all direct channels it knows about

For example, a distributed voting system can be built by combining the two patterns. The vote itself can be triggered simultaneously to direct channels on due time, so that every participants are involved more or less at the same time. And data that is collected in direct channels can ce centralised back to the group channel where results are communicated.

DEFAULT_ADDRESS = 'tcp://127.0.0.1:5555'

check()

Checks configuration settings

This function reads key `bus` and below, and update the context accordingly. It handles following parameters:

- `bus.address` - focal point of bus exchanges on the network. The default value is `tcp://*:5555` which means 'use TCP port 5555 on local machine'.

publish()

Publishes messages

Returns Publisher

Example:

```
# get a publisher for subsequent broadcasts
publisher = bus.publish()

# start the publishing process
publisher.start()

...

# broadcast information_message
publisher.put(channel, message)
```

subscribe(*channels*)

Subcribes to some channels

Parameters **channels** (*str or list of str*) – one or multiple channels

Returns Subscriber

Example:

```
# subscribe from all direct channels related to this group channel
subscriber = bus.subscribe(bot.direct_channels)

...

# get next message from these channels
message = subscriber.get()
```

class shellbot.**Channel**(*attributes=None*)

Bases: object

Represents a chat channel managed by a space

A channel is a group of interactions within a chat space. It features a unique identifier, and a title. It also have participants and content.

This class is a general abstraction of a communication channel, that can easily been adapted to various chat systems. It has been designed as a dictionary wrapper, with minimum exposure to shellbot, while enabling the transmission of rich information through serialization.

Instances of Channel are created within a Space object, and consumed by it as well.

For example, to create a channel with a given title, you could write:

```
channel = space.create(title='A new channel')
bot.say(u"I am happy to join {}".format(channel.title))
```

And to change the title of the channel:

```
channel.title = 'An interesting place'
space.update(channel)
```

Direct channels support one-to-one interactions between the bot and one person. The creation of a direct channel can only be indirect, by sending an invitation to the target person. For example:

```
bot.say(person='foo.bar@acme.com',
        text='Do you want to deal with me?')
```

If the person receives and accepts the invitation, the engine will receive a `join` event and load a new bot devoted to the direct channel. So, at the end of the day, when multiple persons interact with a shellbot, this involve both group and direct channels.

For example, if you create a shellbot named shelly, that interacts with Alice and with Bob, then shelly will overlook multiple bots and channels:

- shelly main channel (bot + channel + store + state machine)
- direct channel with Alice (bot + channel + store + state machine)
- direct channel with Bob (bot + channel + store + state machine)

get(*key*, *default=None*)

Returns the value of one attribute :param key: name of the attribute :type key: str

Parameters **default** (*str or other serializable object*) – default value of the attribute

Returns value of the attribute

Return type str or other serializable object or None

The use case for this function is when you adapt a channel that does not feature an attribute that is expected by shellbot. More specifically, call this function on optional attributes so as to avoid AttributeError

id

Returns channel unique id

Return type str

is_direct

Indicates if this channel is only for one person and the bot

Return type str

A channel is deemed direct when it is reserved to one-to-one interactions. Else it is considered a group channel, with potentially many participants.

is_moderated
Indicates if this channel is moderated

Return type str

A channel is moderated when some participants have specific powers that others do not have. Else all participants are condidered the same and peer with each others.

title
Returns channel title

Return type str

class shellbot.**Command**(*engine=None*, ***kwargs*)
Bases: object

Implements one command

execute(*bot*, *arguments=None*, ***kwargs*)
Executes this command

Parameters

- **bot** (*Shellbot*) – The bot for this execution
- **arguments** (str or None) – The arguments for this command

The function is invoked with a variable number of arguments. Therefore the need for `**kwargs`, so that your code is safe in all cases.

The recommended signature for commands that handle textual arguments is the following:

‘‘‘ def execute(self, bot, arguments=None, **kwargs):

... if arguments:

...

‘‘‘

In this situation, `arguments` contains all text typed after the verb itself. For example, when the command `magic` is invoked with the string:

```
magic rub the lamp
```

then the related command instance is called like this:

```
magic = shell.command('magic')
magic.execute(bot, arguments='rub the lamp')
```

For commands that can handle file attachments, you could use following approach:

```
def execute(self,
            bot,
            arguments=None,
            attachment=None,
            url=None,
            **kwargs):
    ...
    if url:  # a document has been uploaded with this command
```

```
        content = bot.space.download_attachment(url)
        ...
```

Reference information on parameters provided by the shell:

- `bot` - This is the bot instance for which the command is executed. From this you can update the chat with `bot.say()`, or access data attached to the bot in `bot.store`. The engine and all global items can be access with `bot.engine`.
- `arguments` - This is a string that contains everything after the command verb. When `hello How are you doing?` is submitted to the shell, `hello` is the verb, and `How are you doing?` are the arguments. This is the regular case. If there is no command `hello` then the command `*default` is used instead, and arguments provided are the full line `hello How are you doing?`.
- `attachment` - When a file has been uploaded, this attribute provides its external name, e.g., `picture024.png`. This can be used in the executed command, if you keep in mind that the same name can be used multiple times in a conversation.
- `url` - When a file has been uploaded, this is the handle by which actual content can be retrieved. Usually, ask the underlying space to get a local copy of the document.

This function should report on progress by sending messages with one or multiple `bot.say("Whatever response")`.

in_direct = True

in_group = True

information_message = None

is_hidden = False

keyword = None

on_init()

Handles extended initialisation

This function should be expanded in sub-class, where necessary.

Example:

```
def on_init(self):
    self.engine.register('stop', self)
```

usage_message = None

class shellbot.**Context**(*settings=None*, *filter=None*)

Bases: object

Stores settings across multiple independent processing units

This is a key-value store, that supports concurrency across multiple processes.

apply(*settings={}*)

Applies multiple settings at once

Parameters **settings** (*dict*) – variables to be added to this context

check(*key*, *default=None*, *is_mandatory=False*, *validate=None*, *filter=False*)

Checks some settings

Parameters

- **key** – the key that has to be checked
- **default** (*str*) – the default value if no statement can be found
- **is_mandatory** (*bool*) – raise an exception if keys are not found
- **validate** (*callable*) – a function called to validate values before the import
- **filter** (*bool*) – look at the content, and change it eventually

Example:

```
context = Context({
    'spark': {
        'room': 'My preferred room',
        'participants':
            ['alan.droit@azerty.org', 'bob.nard@support.tv'],
        'team': 'Anchor team',
        'token': 'hkNWEtMJNkODk3ZDZLOGQ0OVGlZWU1NmYtyY>',
        'webhook': "http://73a1e282.ngrok.io",
        'weird_token', '$WEIRD_TOKEN',
    }
})

context.check('spark.room', is_mandatory=True)
context.check('spark.team')
context.check('spark.weird_token', filter=True)
```

When a default value is provided, it is used to initialize properly a missing key:

```
context.check('general.switch', 'on')
```

Another usage is to ensure that a key has been set:

```
context.check('spark.room', is_mandatory=True)
```

Additional control can be added with the validation function:

```
context.check('general.switch',
              validate=lambda x: x in ('on', 'off'))
```

When filter is True, if the value is a string starting with '$', then a variable with the same name is loaded from the environment:

```
>>>token=context.check('spark.weird_token', filter=True)
>>>assert token == os.environ.get('WEIRD_TOKEN')
True
```

The default filter can be changed at the creation of a context:

```
>>>context=Context(filter=lambda x : x + '...')
```

This function raises `KeyError` if a mandatory key is absent. If a validation function is provided, then a `ValueError` can be raised as well in some situations.

clear()
Clears content of a context

decrement(*key*, *delta=1*)
Decrements a value

get (*key*, *default=None*)
Retrieves the value of one configurationkey

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
message = context.get('bot.on_start')
```

This function is safe on multiprocessing and multithreading.

has (*prefix*)
Checks the presence of some prefix

Parameters **prefix** (*str*) – key prefix to be checked

Returns True if one or more key start with the prefix, else False

This function looks at keys actually used in this context, and return True if prefix is found. Else it returns False.

Example:

```
context = Context(settings={'space': {'title', 'a title'}})

>>>context.has('space')
True

>>>context.has('space.title')
True

>>>context.has('spark')
False
```

increment (*key*, *delta=1*)
Increments a value

is_empty
Does the context store something?

Returns True if there at least one value, False otherwise

set (*key*, *value*)
Changes the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

Example:

```
context.set('bot.on_start', 'hello world')
```

This function is safe on multiprocessing and multithreading.

classmethod set_logger (*level=10*)
Configure logging

Parameters **level** – expected level of verbosity

This utility function should probably be put elsewhere

class shellbot.**Engine**(*context=None*, *settings={}*, *configure=False*, *mouth=None*, *ears=None*, *fan=None*, *space=None*, *type=None*, *server=None*, *store=None*, *command=None*, *commands=None*, *driver=<class 'shellbot.bot.ShellBot'>*, *machine_factory=None*, *updater_factory=None*, *preload=0*)

Bases: object

Powers multiple bots

The engine manages the infrastructure that is used accross multiple bots acting in multiple spaces. It is made of an extensible set of components that share the same context, that is, configuration settings.

Shellbot allows the creation of bots with a given set of commands. Each bot instance is bonded to a single chat space. The chat space can be either created by the bot itself, or the bot can join an existing space.

The first use case is adapted when a collaboration space is created for semi-automated interactions between human and machines. In the example below, the bot controls the entire life cycle of the chat space. A chat space is created when the program is launched. And it is deleted when the program is stopped.

Example of programmatic chat space creation:

```
from shellbot import Engine, ShellBot, Context, Command
Context.set_logger()

# create a bot and load command
#
class Hello(Command):
    keyword = 'hello'
    information_message = u"Hello, World!"

engine = Engine(command=Hello(), type='spark')

# load configuration
#
engine.configure()

# create a chat space, or connect to an existing one
# settings of the chat space are provided
# in the engine configuration itself
#
engine.bond(reset=True)

# run the engine
#
engine.run()

# delete the chat channel when the engine is stopped
#
engine.dispose()
```

A second interesting use case is when a bot is invited to an existing chat space. On such an event, a new bot instance can be created and bonded to the chat space.

Example of invitation to a chat space:

```
def on_enter(self, channel_id):
    bot = engine.get_bot(channel_id=channel_id)
```

The engine is configured by setting values in the context that is attached to it. This is commonly done by loading the context with a dict before the creation of the engine itself, as in the following example:

```
context = Context({

    'bot': {
        'on_enter': 'You can now chat with Batman',
        'on_exit': 'Batman is now quitting the channel, bye',
    },

    'server': {
        'url': 'http://d9b62df9.ngrok.io',
        'hook': '/hook',
    },

})

engine = Engine(context=context)

engine.configure()
```

Please note that the configuration is checked and actually used on the call `engine.configure()`, rather on the initialisation itself.

When configuration statements have been stored in a separate text file in YAML format, then the engine can be initialised with an empty context, and configuration is loaded afterwards.

Example:

```
engine = Engine()
engine.configure_from_path('/opt/shellbot/my_bot.yaml')
```

When no configuration is provided to the engine, then default settings are considered for the engine itself, and for various components.

For example, for a basic engine interacting in a Cisco Spark channel:

```
engine = Engine(type='spark')
engine.configure()
```

When no indication is provided at all, the engine loads a space of type 'local'.

So, in other terms:

```
engine = Engine()
engine.configure()
```

is strictly equivalent to:

```
engine = Engine('local')
engine.configure()
```

In principle, the configuration of the engine is set once for the full life of the instance. This being said, some settings can be changed globally with the member function *set()*. For example:

```
engine.set('bot.on_banner': 'Hello, I am here to help')
```

DEFAULT_SETTINGS = {'bot': {'banner.content': '$BOT_BANNER_CONTENT', 'on_enter': '$BOT_ON_ENTER', 'on

bond(*title=None*, *reset=False*, *participants=None*, ***kwargs*)
Bonds to a channel

Parameters

- **title** – title of the target channel
- **reset** (*bool*) – if True, delete previous channel and re-create one
- **participants** (*list of str*) – the list of initial participants (optional)

Type title: str

Returns Channel or None

This function creates a channel, or connect to an existing one. If no title is provided, then the generic title configured for the underlying space is used instead.

For example:

```
channel = engine.bond('My crazy channel')
if channel:
    ...
```

Note: this function asks the listener to load a new bot in its cache on successful channel creation or lookup. In other terms, this function can be called safely from any process for the creation of a channel.

build_bot(*id=None*, *driver=<class 'shellbot.bot.ShellBot'>*)
Builds a new bot

Parameters **id** (*str*) – The unique id of the target space

Returns a ShellBot instance, or None

This function receives the id of a chat space, and returns the related bot.

build_machine(*bot*)
Builds a state machine for this bot

Parameters **bot** (ShellBot) – The target bot

Returns a Machine instance, or None

This function receives a bot, and returns a state machine bound to it.

build_store(*channel_id=None*)
Builds a store for this bot

Parameters **channel_id** (*str*) – Identifier of the target chat space

Returns a Store instance, or None

This function receives an identifier, and returns a store bound to it.

build_updater(*id*)
Builds an updater for this channel

Parameters **id** (*str*) – The identifier of an audited channel

Returns an Updater instance, or None

This function receives a bot, and returns a state machine bound to it.

check()
Checks settings of the engine

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key `bot` and below, and update the context accordingly.

Example:

```
context = Context({

    'bot': {
        'on_enter': 'You can now chat with Batman',
        'on_exit': 'Batman is now quitting the channel, bye',
    },

    'server': {
        'url': 'http://d9b62df9.ngrok.io',
        'hook': '/hook',
    },

})
engine = Engine(context=context)
engine.check()
```

configure(*settings={}*)
Checks settings

Parameters **settings** (*dict*) – configuration information

If no settings is provided, and the context is empty, then `self.DEFAULT_SETTINGS` and `self.space.DEFAULT_SETTINGS` are used instead.

configure_from_file(*stream*)
Reads configuration information

Parameters **stream** (`file`) – the handle that contains configuration information

The function loads configuration from the file and from the environment. Port number can be set from the command line.

configure_from_path(*path='settings.yaml'*)
Reads configuration information

Parameters **path** (*str*) – path to the configuration file

The function loads configuration from the file and from the environment. Port number can be set from the command line.

dispatch(*event*, ***kwargs*)
Triggers objects that have registered to some event

Parameters **event** (*str*) – label of the event

Example:

```
def on_bond(self):
    self.dispatch('bond', bot=this_bot)
```

For each registered object, the function will look for a related member function and call it. For example for the event 'bond' it will look for the member function 'on_bond', etc.

Dispatch uses weakref so that it affords the unattended deletion of registered objects.

dispose(*title=None*, ***kwargs*)
Destroys a named channel

Parameters **title** – title of the target channel

Type title: str

enumerate_bots()
Enumerates all bots

get(*key*, *default=None*)
Retrieves the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
message = engine.get('bot.on_start')
```

This function is safe on multiprocessing and multithreading.

get_bot(*channel_id=None*, ***kwargs*)
Gets a bot by id

Parameters **channel_id** (*str*) – The unique id of the target chat space

Returns a bot instance, or None

This function receives the id of a chat space, and returns the related bot.

If no id is provided, then the underlying space is asked to provide with a default channel, as set in overall configuration.

Note: this function should not be called from multiple processes, because this would create one bot per process. Use the function `engine.bond()` for the creation of a new channel.

get_hook()
Provides the hooking function to receive messages from Cisco Spark

hook(*server=None*)
Connects this engine with back-end API

Parameters **server** (Server) – web server to be used

This function adds a route to the provided server, and asks the back-end service to send messages there.

initialize_store(*bot*)
Copies engine settings to the bot store

load_command(**args*, ***kwargs*)
Loads one commands for this bot

This function is a convenient proxy for the underlying shell.

load_commands(**args*, ***kwargs*)
Loads commands for this bot

This function is a convenient proxy for the underlying shell.

name
Retrieves the dynamic name of this bot

Returns The value of `bot.name` key in current context

Return type str

on_build(*bot*)
Extends the building of a new bot instance

Parameters **bot** (ShellBot) – a new bot instance

Provide your own implementation in a sub-class where required.

Example:

```
on_build(self, bot):
    bot.secondary_machine = Input(...)
```

on_enter(*join*)
Bot has been invited to a chat space

Parameters **join** (Join) – The join event received from the chat space

Provide your own implementation in a sub-class where required.

Example:

```
on_enter(self, join):
    mailer.post(u"Invited to {}".format(join.space_title))
```

on_exit(*leave*)
Bot has been kicked off from a chat space

Parameters **leave** (Leave) – The leave event received from the chat space

Provide your own implementation in a sub-class where required.

Example:

```
on_exit(self, leave):
    mailer.post(u"Kicked off from {}".format(leave.space_title))
```

on_start()
Does additional stuff when the engine is started

Provide your own implementation in a sub-class where required.

on_stop()
Does additional stuff when the engine is stopped

Provide your own implementation in a sub-class where required.

Note that some processes may have been killed at the moment of this function call. This is likely to happen when end-user hits Ctl-C on the keyboard for example.

register(*event*, *instance*)
Registers an object to process an event

Parameters

- **event** (*str*) – label, such as 'start' or 'bond'
- **instance** (*object*) – an object that will handle the event

This function is used to propagate events to any module that may need it via callbacks.

On each event, the engine will look for a related member function in the target instance and call it. For example for the event 'start' it will look for the member function 'on_start', etc.

Following standard events can be registered:

•'bond' - when the bot has connected to a chat channel

- 'dispose' - when resources, including chat space, will be destroyed
- 'start' - when the engine is started
- 'stop' - when the engine is stopped
- 'join' - when a person is joining a space
- 'leave' - when a person is leaving a space

Example:

```
def on_init(self):
    self.engine.register('bond', self)  # call self.on_bond()
    self.engine.register('dispose', self) # call self.on_dispose()
```

If the function is called with an unknown label, then a new list of registered callbacks will be created for this event. Therefore the engine can be used for the dispatching of any custom event.

Example:

```
self.engine.register('input', processor)  # for processor.on_input()
...
received = 'a line of text'
self.engine.dispatch('input', received)
```

Registration uses weakref so that it affords the unattended deletion of registered objects.

run (*server=None*)

Runs the engine

Parameters **server** (Server) – a web server

If a server is provided, it is ran in the background. A server could also have been provided during initialisation, or loaded during configuration check.

If no server instance is available, a loop is started to fetch messages in the background.

In both cases, this function does not return, except on interrupt.

set (*key*, *value*)

Changes the value of one configuration key

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

Example:

```
engine.set('bot.on_start', 'hello world')
```

This function is safe on multiprocessing and multithreading.

start ()

Starts the engine

start_processes ()

Starts the engine processes

This function starts a separate process for each main component of the architecture: listener, speaker, etc.

stop()
Stops the engine

This function changes in the context a specific key that is monitored by bot components.

version
Retrieves the version of this bot

Returns The value of `bot.version` key in current context

Return type str

class shellbot.**Listener**(*engine=None*, *filter=None*)
Bases: multiprocessing.process.Process

Handles messages received from chat space

DEFER_DURATION = 2.0

EMPTY_DELAY = 0.005

FRESH_DURATION = 0.5

idle()
Finds something smart to do

on_inbound(*received*)
Another event has been received

Parameters **received** (*Event or derivative*) – the event received

Received information is transmitted to registered callbacks on the `inbound` at the engine level.

on_join(*received*)
A person, or the bot, has joined a space

Parameters **received** (Join) – the event received

Received information is transmitted to registered callbacks on the `join` at the engine level.

In the special case where the bot itself is joining a channel by invitation, then the event `enter` is dispatched instead.

on_leave(*received*)
A person, or the bot, has left a space

Parameters **received** (Leave) – the event received

Received information is transmitted to registered callbacks on the `leave` at the engine level.

In the special case where the bot itself has been kicked off from a channel, then the event `exit` is dispatched instead.

on_message(*received*)
A message has been received

Parameters **received** (Message) – the message received

Received information is transmitted to registered callbacks on the `message` event at the engine level.

When a message is directed to the bot it is submitted directly to the shell. This is handled as a command, that can be executed immediately, or pushed to the inbox and processed by the worker when possible.

All other input is thrown away, except if there is some downwards listeners. In that situation the input is pushed to a queue so that some process can pick it up and process it.

The protocol for downwards listeners works like this:

- Check the `bot.fan` queue frequently
- On each check, update the string `fan.<channel_id>` in the context with the value of `time.time()`. This will say that you are around.

The value of `fan.<channel_id>` is checked on every message that is not for the bot itself. If this is fresh enough, then data is put to the `bot.fan` queue. Else message is just thrown away.

process(*item*)

Processes items received from the chat space

Parameters **item** (*dict or json-encoded string*) – the item received

This function dispatches items based on their type. The type is a key of the provided dict.

Following types are handled:

- `message` – This is a textual message, maybe with a file attached. The message is given to the `on_message()` function.
- `join` – This is when a person or the bot joins a space. The function `on_join()` is called, providing details on the person or the bot who joined
- `leave` – This is when a person or the bot leaves a space. The function `on_leave()` is called with details on the leaving person or bot.
- `load_bot` – This is a special event to load the cache in the process that is running the listener. The identifier of the channel to load is provided as well.
- on any other case, the function `on_inbound()` is called.

run()

Continuously receives updates

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
listener = Listener(engine=my_engine)
process = listener.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when a poison pill is pushed to the queue. For example:

```
engine.ears.put(None)
```

class shellbot.**MachineFactory**(*module=None*, *name=None*, ***kwargs*)

Bases: `object`

Provides new state machines

In simple situations, you can rely on standard machines, and provide any parameters by these. For example:

```
factory = MachineFactory(module='shellbot.machines.input'
                         question="What's Up, Doc?")

...
```

```
machine = factory.get_machine()
```

When you provide different state machines for direct channels and for group channels, overlay member functions as in this example:

```
class GreatMachineForDirectChannel(Machine):
    ...

class MachineOnlyForGroup(Machine):
    ...

class MyFactory(MachineFactory):

    def get_machine_for_direct_channel(self, bot):
        return GreatMachineForDirectChannel( ... )

    def get_machine_for_group_channel(self, bot):
        return MachineOnlyForGroup( ... )
```

get_default_machine(*bot*)

Gets a new state machine

Parameters **bot** (ShellBot) – The bot associated with this state machine

Example:

```
my_machine = factory.get_default_machine(bot=my_bot)
my_machine.start()
```

This function can be overlaid in a subclass for adapting the production of state machines for default case.

get_machine(*bot=None*)

Gets a new state machine

Parameters **bot** (ShellBot) – The bot associated with this state machine

Example:

```
my_machine = factory.get_machine(bot=my_bot)
my_machine.start()
```

This function detects the kind of channel that is associated with this bot, and provides a suitable state machine.

get_machine_for_direct_channel(*bot*)

Gets a new state machine for a direct channel

Parameters **bot** (ShellBot) – The bot associated with this state machine

Example:

```
my_machine = factory.get_machine_for_direct_channel(bot=my_bot)
my_machine.start()
```

This function can be overlaid in a subclass for adapting the production of state machines for direct channels.

get_machine_for_group_channel(*bot*)

Gets a new state machine for a group channel

Parameters **bot** (ShellBot) – The bot associated with this state machine

Example:

```
my_machine = factory.get_machine_for_group_channel(bot=my_bot)
my_machine.start()
```

This function can be overlaid in a subclass for adapting the production of state machines for group channels.

get_machine_from_class(*bot*, *module*, *name*, ***kwargs*)

Gets a new state machine from a module

Parameters

- **bot** (ShellBot) – The bot associated with this state machine
- **module** (*str*) – The python module to import
- **name** (*str*) – The class name to instantiate (optional)

Example:

```
machine = factory.get_machine_from_class(my_bot,
                                         'shellbot.machines.base',
                                         'Machine')
```

class shellbot.**Notifier**(*context=None*, ***kwargs*)

Bases: *shellbot.routes.base.Route*

Notifies a queue on web request

>>>queue = Queue() >>>route = Notifier(route='/notify', queue=queue, notification='hello')

When the route is requested over the web, the notification is pushed to the queue.

>>>queue.get() 'hello'

Notification is triggered on GET, POST, PUT and DELETE verbs.

delete()

get(***kwargs*)

notification = None

notify()

post()

put()

queue = <shellbot.routes.notifier.NoQueue object>

route = '/notify'

class shellbot.**Publisher**(*context*)

Bases: multiprocessing.process.Process

Publishes asynchronous messages

For example, from a group channel, you may send instructions to every direct channels:

```
# get a publisher
publisher = bus.publish()
```

```
# send instruction to direct channels
publisher.put(bot.direct_channels, instruction)
```

From within a direct channel, you may reflect your state to observers:

```
# get a publisher
publish = bus.publish()

# share new state
publisher.put(bot.id, bit_of_information_here)
```

DEFER_DURATION = 0.3

EMPTY_DELAY = 0.005

process(*item*)

Processes items received from the queue

Parameters **item** (*str*) – the item received

Note that the item should result from serialization of (channel, message) tuple done previously.

put(*channels*, *message*)

Broadcasts a message

Parameters

- **channels** (*str or list of str*) – one or multiple channels
- **message** (*dict or other json-serializable object*) – the message to send

Example:

```
message = { ... }
publisher.put(bot.id, message)
```

This function actually put the message in a global queue that is handled asynchronously. Therefore, when the function returns there is no guarantee that message has been transmitted nor received.

run()

Continuously broadcasts messages

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
publisher = Publisher(address)
process = publisher.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when a poison pill is pushed to the queue. For example:

```
publisher.fan.put(None)
```

class shellbot.**Route**(*context=None*, ***kwargs*)
Bases: object

Implements one route

delete()

get(***kwargs*)

post()

put()

route = None

class shellbot.**Server**(*context=None*, *httpd=None*, *route=None*, *routes=None*, *check=False*)
Bases: bottle.Bottle

Serves web requests

add_route(*item*)
Adds one web route

Parameters **route** (Route) – one additional route

add_routes(*items*)
Adds web routes

Parameters **routes** (*list of routes*) – a list of additional routes

configure(*settings={}*)
Checks settings of the server

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key server and below, and update the context accordingly:

```
>>>server.configure({'server': {
       'binding': '10.4.2.5',
       'port': 5000,
       'debug': True,
       }})
```

This can also be written in a more compact form:

```
>>>server.configure({'server.port': 5000})
```

route(*route*)
Gets one route by path

Returns the related route, or None

routes
Lists all routes

Returns a list of routes, or []

Example:

```
>>>server.get_routes()
['/hello', '/world']
```

run()
Serves requests

class shellbot.**Shell**(*engine*)

Bases: object

Parses input and reacts accordingly

command(*keyword*)

Get one command

Parameters **keyword** (*str*) – the keyword for this command

Returns the instance for this command

Return type *command* or None

Lists available commands and related usage information.

Example:

```
>>>print(shell.command('help').information_message)
```

commands

Lists available commands

Returns a list of verbs

Return type list of str

This function provides with a dynamic inventory of all capabilities of this shell.

Example:

```
>>>print(shell.commands)
['*default', '*empty', 'help']
```

configure(*settings={}*)

Checks settings of the shell

Parameters **settings** (*dict*) – a dictionary with some statements for this instance

This function reads key `shell` and below, and update the context accordingly:

```
>>>shell.configure({'shell': {
       'commands':
          ['examples.exception.state', 'examples.exception.next']
       }})
```

This can also be written in a more compact form:

```
>>>shell.configure({'shell.commands':
       ['examples.exception.state', 'examples.exception.next']
       })
```

Note that this function does preserve commands that could have been loaded previously.

do(*line*, *received=None*)

Handles one line of text

Parameters

- **line** (*str*) – a line of text to parse and to handle
- **received** (Message) – the message that contains the command

This function uses the first token as a verb, and looks for a command of the same name in the shell.

If the command does not exist, the command `*default` is used instead. Default behavior is implemented in `shellbot.commands.default` yet you can load a different command for customization.

If an empty line is provided, the command `*empty` is triggered. Default implementation is provided in `shellbot.commands.empty`.

When a file has been uploaded, the information is given to the command that is executed. If no message is provided with the file, the command `*upload` is triggered instad of `*empty`. Default implementation is provided in `shellbot.commands.upload`.

Following parameters are used for the execution of a command:

- `bot` - A bot instance is retrieved from the channel id mentioned in `received`, and provided to the command.
- `arguments` - This is a string that contains everything after the command verb. When `hello How are you doing?` is submitted to the shell, `hello` is the verb, and `How are you doing?` are the arguments. This is the regular case. If there is no command `hello` then the command `*default` is used instead, and arguments provided are the full line `hello How are you doing?`.
- `attachment` - When a file has been uploaded, this attribute provides its external name, e.g., `picture024.png`. This can be used in the executed command, if you keep in mind that the same name can be used multiple times in a conversation.
- `url` - When a file has been uploaded, this is the handle by which actual content can be retrieved. Usually, ask the underlying space to get a local copy of the document.

load_command(*command*)

Loads one command for this shell

Parameters **command** (*str or command*) – A command to load

If a string is provided, it should reference a python module that can be used as a command. Check `base.py` in `shellbot.commands` for a clear view of what it means to be a vaid command for this shell.

Example:

```
>>>shell.load_command('shellbot.commands.help')
```

If an object is provided, it should duck type the command defined in `base.py` in `shellbot.commands`.

Example:

```
>>>from shellbot.commands.version import Version
>>>command = Version()
>>>shell.load_command(command)
```

load_commands(*commands=[]*)

Loads commands for this shell

Parameters **commands** (*List of labels or list of commands*) – A list of commands to load

Example:

```
>>>commands = ['shellbot.commands.help']
>>>shell.load_commands(commands)
```

Each label should reference a python module that can be used as a command. Check `base.py` in `shellbot.commands` for a clear view of what it means to be a vaid command for this shell.

If objects are provided, they should duck type the command defined in `base.py` in `shellbot.commands`.

Example:

```
>>>from shellbot.commands.version import Version
>>>version = Version()
>>>from shellbot.commands.help import Help
>>>help = Help()
>>>shell.load_commands([version, help])
```

load_default_commands()

Loads default commands for this shell

Example:

```
>>>shell.load_default_commands()
```

class shellbot.**ShellBot**(*engine*, *channel_id=None*, *space=None*, *store=None*, *fan=None*, *machine=None*)

Bases: object

Manages interactions with one space, one store, one state machine

A bot consists of multiple components devoted to one chat channel: - a space - a store - a state machine - ... other optional components that may prove useful

It is designated by a unique id, that is also the unique id of the channel itself.

A bot relies on an underlying engine instance for actual access to the infrastructure, including configuration settings.

The life cycle of a bot can be described as follows:

1.A bot is commonly created from the engine, or directly:

```
bot = ShellBot(engine, channel_id='123')
```

2.The space is connected to some back-end API:

```
space.connect()
```

3.Multiple channels can be handled by a single space:

```
channel = space.create(title)

channel = space.get_by_title(title)
channel = space.get_by_id(id)

channel.title = 'A new title'
space.update(channel)

space.delete(id)
```

Channels feature common attributes, yet can be extended to convey specificities of some platforms.

4.Messages can be posted:

```
space.post_message(id, 'Hello, World!')
```

5.The interface allows for the addition or removal of channel participants:

```
space.add_participants(id, persons)
space.add_participant(id, person, is_moderator)
space.remove_participants(id, persons)
space.remove_participant(id, person)
```

add_participant(*person*, *is_moderator=False*)
Adds one participant

Parameters **person** (*str*) – e-mail addresses of person to add

The underlying platform may, or not, take the optional parameter `is_moderator` into account. The default bahaviour is to discard it, as if the parameter had the value `False`.

add_participants(*persons=[]*)
Adds multiple participants

Parameters **persons** (*list of str*) – e-mail addresses of persons to add

append(*key*, *item*)
Appends an item to a list

Parameters

- **key** (*str*) – name of the list
- **item** (*any serializable type is accepted*) – a new item to append

Example:

```
>>>bot.append('names', 'Alice')
>>>bot.append('names', 'Bob')
>>>bot.recall('names')
['Alice', 'Bob']
```

bond()
Bonds to a channel

This function is called either after the creation of a new channel, or when the bot has been invited to an existing channel. In such situations the banner should be displayed as well.

There are also situations where the engine has been completely restarted. The bot bonds to a channel where it has been before. In that case the banner should be avoided.

dispose(***kwargs*)
Disposes all resources

This function deletes the underlying channel in the cloud and resets this instance. It is useful to restart a clean environment.

>>>bot.bond(title="Working Space") ... >>>bot.dispose()

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

forget(*key=None*)
Forgets a value or all values

Parameters **key** (*str*) – name of the value to forget, or None

To clear only one value, provides the name of it. For example:

```
bot.forget('variable_123')
```

To clear all values in the store, just call the function without a value. For example:

```
bot.forget()
```

id
Gets unique id of the related chat channel

Returns the id of the underlying channel, or None

is_ready
Checks if this bot is ready for interactions

Returns True or False

on_bond()
Adds processing to channel bonding

This function should be changed in sub-class, where necessary.

Example:

```
def on_bond(self):
    do_something_important_on_bond()
```

on_enter()
Enters a channel

on_exit()
Exits a channel

on_init()
Adds to bot initialization

It can be overlaid in subclass, where needed

on_reset()
Adds processing to space reset

This function should be expanded in sub-class, where necessary.

Example:

```
def on_reset(self):
    self._last_message_id = 0
```

recall(*key*, *default=None*)
Recalls a value

Parameters

- **key** (*str*) – name of the value
- **default** (*any serializable type is accepted*) – default value

Returns the actual value, or the default value, or None

Example:

```
value = bot.recall('variable_123')
```

remember (*key*, *value*)
Remembers a value

Parameters

- **key** (*str*) – name of the value
- **value** (*any serializable type is accepted*) – new value

This functions stores or updates a value in the back-end storage system.

Example:

```
bot.remember('variable_123', 'George')
```

remove_participant (*person*)
Removes one participant

Parameters **person** (*str*) – e-mail addresses of person to add

remove_participants (*persons=[]*)
Removes multiple participants

Parameters **persons** (*list of str*) – e-mail addresses of persons to remove

reset ()
Resets a space

After a call to this function, `bond()` has to be invoked to return to normal mode of operation.

say (*text=None*, *content=None*, *file=None*, *person=None*)
Sends a message to the chat space

Parameters

- **text** (*str or None*) – Plain text message
- **content** (*str or None*) – Rich content such as Markdown or HTML
- **file** (*str or None*) – path or URL to a file to attach
- **person** (*str*) – for direct message to someone

say_banner ()
Sends banner to the channel

This function uses following settings from the context:

- `bot.banner.text` or `bot.on_enter` - a textual message
- `bot.banner.content` - some rich content, e.g., Markdown or HTML
- `bot.banner.file` - a document to be uploaded

The quickest setup is to change `bot.on_enter` in settings, or the environment variable `$BOT_ON_ENTER`.

Example:

```
os.environ['BOT_ON_ENTER'] = 'You can now chat with Batman'
engine.configure()
```

Then there are situtations where you want a lot more flexibility, and rely on a smart banner. For example you could do the following:

```
settings = {
    'bot': {
        'banner': {
            'text': u"Type '@{} help' for more information",
            'content': u"Type ``@{} help`` for more information",
            'file': "http://on.line.doc/guide.pdf"
        }
    }
}

engine.configure(settings)
```

When bonding to a channel, the bot will send an update similar to the following one, with a nice looking message and image:

```
Type '@Shelly help' for more information
```

Default settings for the banner rely on the environment, so it is easy to inject strings from the outside. Use following variables:

- `$BOT_BANNER_TEXT` or `$BOT.ON_ENTER` - the textual message
- `$BOT_BANNER_CONTENT` - some rich content, e.g., Markdown or HTML
- `$BOT_BANNER_FILE` - a document to be uploaded

title

Gets title of the related chat channel

Returns the title of the underlying channel, or None

update(*key*, *label*, *item*)

Updates a dict

Parameters

- **key** (*str*) – name of the dict
- **label** (*str*) – named entry in the dict
- **item** (*any serializable type is accepted*) – new value of this entry

Example:

```
>>>bot.update('input', 'PO Number', '1234A')
>>>bot.update('input', 'description', 'some description')
>>>bot.recall('input')
{'PO Number': '1234A',
 'description': 'some description'}
```

class shellbot.**SpaceFactory**

Bases: object

Builds a space from configuration

Example:

```
my_context = Context(settings={
    'space': {
        'type': "spark",
        'room': 'My preferred room',
        'participants':
```

```
            ['alan.droit@azerty.org', 'bob.nard@support.tv'],
        'team': 'Anchor team',
        'token': 'hkNWEtMJNkODk3ZDZLOGQ0OVGlZWU1NmYtyY',
        'fuzzy_token': '$MY_FUZZY_SPARK_TOKEN',
    }
})

space = SpaceFactory.build(context=my_context)
```

classmethod **build**(*context*, ***kwargs*)

Builds an instance based on provided configuration

Parameters **context** (Context) – configuration to be used

Returns a ready-to-use space

Return type *Space*

This function "senses" for a type in the context itself, then provides with an instantiated object of this type.

A ValueError is raised when no type can be identified.

classmethod **get**(*type*, ***kwargs*)

Loads a space by type

Parameters **type** (*str*) – the required space

Returns a space instance

This function seeks for a suitable space class in the library, and returns an instance of it.

Example:

```
space = SpaceFactory.get('spark', ex_token='123')
```

A ValueError is raised if the type is unknown.

classmethod **sense**(*context*)

Detects type from configuration

Parameters **context** (Context) – configuration to be analyzed

Returns a guessed type

Return type str

Example:

```
type = SpaceFactory.sense(context)
```

A ValueError is raised if no type could be identified.

types = {'spark': <class 'shellbot.spaces.ciscospark.SparkSpace'>, 'local': <class 'shellbot.spaces.local.LocalSpace'>, 'sp

class shellbot.**Speaker**(*engine=None*)

Bases: multiprocessing.process.Process

Sends updates to a business messaging space

EMPTY_DELAY = 0.005

process(*item*)

Sends one update to a business messaging space

Parameters **item** (*str or object*) – the update to be transmitted

run()

Continuously send updates

This function is looping on items received from the queue, and is handling them one by one in the background.

Processing should be handled in a separate background process, like in the following example:

```
speaker = Speaker(engine=my_engine)
speaker.start()
```

The recommended way for stopping the process is to change the parameter `general.switch` in the context. For example:

```
engine.set('general.switch', 'off')
```

Alternatively, the loop is also broken when an exception is pushed to the queue. For example:

```
engine.mouth.put(None)
```

class shellbot.**Subscriber**(*context*, *channels*)

Bases: object

Subscribes to asynchronous messages

For example, from a group channel, you may subscribe from direct channels of all participants:

```
# subscribe from all direct channels related to this group channel
subscriber = bus.subscribe(bot.direct_channels)

# get messages from direct channels
while True:
    message = subscriber.get()
    ...
```

From within a direct channel, you may receive instructions sent by the group channel:

```
# subscribe for messages sent to me
subscriber = bus.subscribe(bot.id)

# get and process instructions one at a time
while True:
    instruction = subscriber.get()
    ...
```

get(*block=False*)

Gets next message

Returns dict or other serializable message or None

This function returns next message that has been made available, or None if no message has arrived yet.

Example:

```
message = subscriber.get()  # immedaite return
if message:
    ...
```

Change the parameter `block` if you prefer to wait until next message arrives.

Example:

```
message = subscriber.get(block=True)  # wait until available
```

Note that this function does not preserve the enveloppe of the message. In other terms, the channel used for the communication is lost in translation. Therefore the need to put within messages all information that may be relevant for the receiver.

class `shellbot.`**`Wrapper`**(*context=None*, ***kwargs*)

Bases: *shellbot.routes.base.Route*

Calls a function on web request

When the route is requested over the web, the wrapped function is called.

Example:

```
def my_callable(**kwargs):
    ...

route = Wrapper(callable=my_callable, route='/hook')
```

Wrapping is triggered on GET, POST, PUT and DELETE verbs.

`callable` = None

`delete`()

`get`(***kwargs*)

`post`()

`put`()

`route` = None

CHAPTER 7

History

17.8.5

- Add tutorials to the on-line documentation (based on examples)
- Bot can now be invited to any direct or group channel by end-user
- Complete review of the internal design of processes, etc.
- Major revamp of the code
- Travis-CI passed on python 2.7 and 3.5
- 94% test coverage
- 3094 python statements

17.6.6

- Add example Hotel California to demonstrate automated join and leave events
- Add example Escalation to sum up all capabilities of shellbot
- Expand the set of real-time events managed by shellbot: ‘join’, ‘leave’
- Protect contexts and stores from KeyboardInterrupt
- Minor improvements on code and tests
- Travis-CI passed on python 2.7 and 3.5
- 94% test coverage
- 2236 python statements

17.5.28

- Fix examples
- Minor improvements on code and tests
- Travis-CI passed on python 2.7 and 3.5
- 95% test coverage
- 2130 python statements

17.5.27

- Fix package content
- Add events 'start' and 'stop' to bot
- Use weakref with bot event listeners
- Minor improvements on code and tests
- Travis-CI passed on python 2.7
- 95% test coverage
- 2128 python statements

17.5.22

- Full mirroring of chat and files in a secondary Cisco Spark room
- Add updater to Elasticsearch for chat indexing
- Add updater to file system (useful for archiving chat interactions)
- Download attachments from a Cisco Spark room
- Add callbacks at bot level
- Abstract inbound events from chat space
- Minor improvements on code and tests
- Travis-CI passed on python 2.7 and 3.5
- 94% test coverage
- 2118 python statements

17.5.16

- Add permanent thread-safe storage to bot (Sqlite as first store)
- Introduce updaters as flexible mechanism to replicate input messages
- Add example to demonstrate chat audit in a secondary room
- Minor improvements on code and tests

- Force garbage collection in heavy tests
- Travis-CI passed on python 2.7 and 3.5
- 93% test coverage
- 1784 python statements

17.5.7

- Add example to demonstrate interactive capture of data
- Add example to demonstrate chat simulator
- Add serious state machine and use it for asynchronous input
- Add fittings plan for automated deployment on MCP with plumbery
- Early development of a new command to audit chats
- Minor improvements on code and tests
- Travis-CI passed on python 2.7 and 3.5
- 93% test coverage
- 1514 python statements

17.5.2

- Add example 'notify' to ease introductory use case
- Minor improvements on code and tests
- Travis-CI passed on python 2.7 and 3.5
- 91% test coverage
- 1117 python statements

17.4.28

- Fix the build of ReadTheDocs documentation
- Expand explanations within example scripts
- Implement default bot configuration with named environment variables
- Travis-CI passed on python 2.7 and 3.5
- 91% test coverage
- 1096 python statements

17.4.27

- Examples: hello, batman, pushy, todos, buzz
- Initial push of API docs
- Travis-CI passed on python 2.7 and 3.5
- 91% test coverage
- 1094 python statements

17.4.18

- Initial push to PyPi
- Travis-CI passed on python 2.7 and 3.5
- 81% test coverage
- 932 python statements

17.4.03

- Initial push to GitHub - no release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W