DevAssistant Documentation

Release 0.10.0

Bohuslav Kabrda, Petr Hracek

November 13, 2014

Contents

DevAssistant - making life easier for developers

CHAPTER 1

Contents

1.1 User Documentation

1.1.1 A Brief Intro

DevAssistant - start developing with ease

DevAssistant (http://devassistant.org) can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc.

It is based on idea of per-{language/framework/...} "assistants" (plugins) with hierarchical structure.

Note: prior to version 0.10.0, DevAssistant has been shipped with a default set of assistants that only worked on Fedora. We decided to drop this default set and create DAPI, DevAssistant Package Index, https://dapi.devassistant.org/ *- an upstream PyPI/Rubygems-like repository of packaged assistants. DAPI's main aim is to create a community around DevAssistant and provide various assistants with good support for various platforms - a task that DevAssistant core team alone is not able to achieve for a large set of assistants.*

This all means that if you get DevAssistant from upstream repo or from PyPI, you will have no assistants installed by default. To get assistants, search DAPI through web browser or run `da pkg search <term>` *and* `da pkg install <assistant package>`*. This will install one or more DAPs (DevAssistant Packages) with the desired assistants.*

If you want to create your own assistants and upload them to DAPI, see http://docs.devassistant.org/en/latest/developer_documentation/create_assistant.html and http://docs.devassistant.org/en/latest/developer_documentation/create_assistant/packaging_and_distributing.html.

There are four main modes of DevAssistant execution. Explanations are provided to better illustrate what each mode is supposed to do:

`create` Create new projects - scaffold source code, install dependencies, initialize SCM repos ...

`tweak` Work with existing projects - add source files, import to IDEs, push to GitHub, ...

`prepare` Prepare environment for working with existing upstream projects - install dependencies, set up services, ...

`extras` Tasks not related to a specific project, e.g. enabling services, setting up IDEs, ...

These are some examples of what you can do:

```
# search for assistants that have "Django" in their description
$ da pkg search django
python - Python assistants (library, Django, Flask, GTK3)

# install the found "python" DAP, assuming it supports your OS/distro
```

```
$ da pkg install python

# find out if the installed package has documentation
$ da doc python
INFO: DAP "python" has these docs:
...
INFO: usage.txt
...
# show help
$ da doc python usage.txt

# if the documentation doesn't say it specifically, find out if there is a "create"
#  assistant in the installed "python" DAP
$ da create -h
...
{..., python, ...}
...

# there is, so let's find out if it has any subassistants
$ da create python -h
...
{..., django, ...}
...

# we found out that there is "django" subassistant, let's find out how to use it
$ da create python django -h
<help text with commandline options>

# help text tells us that django assistant doesn't have subassistants and is runnable, let's do it
$ da create python django -n ~/myproject # sets up Django project named "myproject" inside your home

# using the same approach with "pkg search", "pkg install" and "da tweak -h",
#  we find, install and read help for "tweak" assistant that imports projects to eclipse
$ da tweak eclipse -p ~/myproject # run in project dir or use -p to specify path

# using the same approach, we find, install and read help for assistant
#  that tries to prepare environment for a custom upstream project, possibly utilizing
#  its ".devassistant" file
$ da prepare custom -u scm_url -p directory_to_save_to

# sometimes, DevAssistant can really do a very special thing for you ...
$ da extras make-coffee
```

Should you have some questions, feel free to ask us at Freenode channel #devassistant or on our mailing list (https://lists.fedoraproject.org/mailman/listinfo/devassistant). You can also join our G+ community (https://plus.google.com/u/0/communities/112692240128429771916) or follow us on Twitter (https://twitter.com/dev_assistant).

1.1.2 So What is an Assistant?

In short, assistant is a recipe for creating/tweaking a project or setting up the environment in a certain way. DevAssistant is in fact just a core that "runs" assistants according to certain rules.

Each assistant specifies a way to achieve a single task, e.g. create a new project in framework X of language Y.

If you want to know more about how this all works, consult *Create Your Own Assistant*.

Assistant Roles

There are four assistant roles:

creator (`create` or `crt` on command line) creates new projects

tweak (`tweak` or `twk` on command line) works with existing projects

preparer (`prepare` or `prep` on command line) prepares environment for development of upstream projects

extras (`extras` or `extra` on command line) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

You can learn about how to invoke the respective roles below in *Creating New Projects*, *Working with Existing Projects*, *Preparing Environment* and *Extras*.

1.1.3 Using Commandline Interface

Creating New Projects

DevAssistant can help you create your projects with one line in a terminal. For example:

```
$ da create python django -n foo -e -g
```

`da` is the short form of `devassistant`. You can use either of them, but `da` is preferred.

What this line does precisely depends on the author of the assistant. You can always display help by using `da create python django -h`. Running the above command line *may* do something like this:

- Install Django and all needed dependencies.
- Create a Django project named `foo` in the current working directory.
- Make any necessary adjustments so that you can run the project and start developing right away.
- The `-e` switch will make DevAssistant register the newly created projects into Eclipse. This will also cause installation of Eclipse and PyDev, unless already installed.
- The `-g` switch will make DevAssistant register the project on Github and push sources there.

Working with Existing Projects

DevAssistant allows you to work with previously created projects. You can do this by using `da tweak`, as opposed to `da create` for creating:

```
$ da tweak eclipse
```

As noted above, what an assistant does depends on its author. In this case, it seems that the assistant will import an existing project into Eclipse, possibly installing missing dependencies - to find out if this assumption is correct, run `da tweak eclipse -h` and read the help.

Preparing Environment

DevAssistant can set up the environment and install dependencies for development of an already existing project located in a remote SCM (e.g. Github). There is, for example, the so-called custom prepare assistant, that is supposed to prepare environment for arbitrary upstream projects. This means that it will checkout the source code from given git

repo and if there is a `.devassistant` file in the repo, it'll install dependencies and prepare environment according to it:

```
$ da prepare custom -u scm_url
```

Warning: The `custom` assistant executes custom pieces of code from a `.devassistant` file, so use this only for projects whose upstreams you trust.

We hope that existance of DAPI will attract people from various upstreams to create prepare assistants for their specific projects, so that people could do something like:

```
$ da prepare openstack
```

To get development environment prepared for development of OpenStack, etc...

Extras

The last piece of functionality is performing arbitrary tasks that are not related to a specific projects. E.g.:

```
$ da extras make-coffee
```

Custom Actions

There are also some custom actions besides `create`, `tweak`, `prepare` and `extras`.

- `doc` - Displays documentation for given DAP. Uses `less` as pager, if available.:

  ```
  # finds out if "python" DAP has documentation, lists documents if yes
  $ da doc python
  ...
  INFO: LICENSE
  INFO: somedoc.txt
  INFO: docsubdir/someotherdoc.txt
  ...

  # displays specific document for "python" DAP
  $ da doc python docsubdir/someotherdoc.txt
  ```

- `help`- Displays help :)
- `pkg` - Manipulate dap packages, communicate with DAPI. Has several subactions:
  - `info` - prints information about packages from DAPI
  - `install` - installs packages from DAPI
  - `lint` - runs sanity checks on local DAP package
  - `list` - lists installed DAPs
  - `search` - searches DAPs on DAPI for given term
  - `uninstall` - uninstalls given package(s)
  - `update` - updates all or given package(s)
- `version`- Displays current DevAssistant version.

1.1.4 Using the GUI

The DevAssistant GUI provides the full functionality of *Commandline Interface* through a Gtk based application.

The GUI provides all assistant of the same type (creating, tweaking, preparing and extras) in one tab to keep things organized.

The GUI workflow is dead simple:

- Choose the assistant that you want to use, click it and possibly choose a proper subassistant (e.g. `django` for `python`).
- The GUI displays a window where you can modify some settings and choose from various assistant-specific options.
- Click the "Run" button and then just watch getting the stuff done. If your input is needed (such as confirming dependencies to install), DevAssistant will ask you, so don't go get your coffee just yet.
- After all is done, get your coffee and enjoy.

1.2 Developer Documentation

1.2.1 DevAssistant Core

Note: So far, this only covers some bits and pieces of the whole core.

DevAssistant Load Paths

DevAssistant has couple of load path entries, that are searched for assistants, snippets, icons and files used by assistants. In standard installations, there are three paths:

1. "user" path, `~/.devassistant/`
2. "local" path, `/usr/local/share/devassistant/`
3. "system" path, `/usr/share/devassistant/`

Another path(s) can be added by specifying `DEVASSISTANT_PATH` environment variable (if more paths are used, they must be separated by colon). These paths are prepended to the list of standard load paths.

Each load path entry has this structure:

```
assistants/
  crt/
  twk/
  prep/
  extra/
files/
  crt/
  twk/
  prep/
  extra/
  snippets/
icons/
  crt/
  twk/
  prep/
```

```
  extra/
snippets/
```

Icons under `icons` directory and files in `files` directory "copy" must the structure of `assistants` directory. E.g. for assistant `assistants/crt/foo/bar.yaml`, the icon must be `icons/crt/foo/bar.svg` and files must be placed under `files/crt/foo/bar/`

Assistants Loading Mechanism

DevAssistant loads assistants from all load paths mentioned above (more specifically from `<load_path>/assistants/` only), traversing them in order "user", "local", "system".

When DevAssistant starts up, it loads all assistants from all these paths. It assumes, that Creator assistants are located under `crt` subdirectories the same applies to Tweak (`twk`), Preparer (`prep`) and Extras (`extra`) assistants.

For example, loading process for Creator assistants looks like this:

1. Load all assistants located in `crt` subdirectories of each `<load path>/assistants/` (do not descend into subdirectories). If there are multiple assistants with the same name in different load paths, the first traversed wins.
2. For each assistant named `foo.yaml`:
   (a) If `crt/foo` directory doesn't exist in any load path entry, then this assistant is "leaf" and therefore can be directly used by users.
   (b) Else this assistant is not leaf and DevAssistant loads its subassistants from the directory, recursively going from point 1).

Command Runners

Command runners... well, they run commands. They are the functionality that makes DevAssistant powerful, since they effectively allow you to create callbacks to Python, where you can cope with the hard parts unsuitable for Yaml assistants.

When DevAssistant executes a `run` section, it reads commands one by one and dispatches them to their respective command runners. Every command runner can do whatever it wants - for example, we have a command runner that creates Github repos.

After a command runner is run, DevAssistant sets `LAST_LRES` and `LAST_RES` global variables for usage (these are rewritten with every command run). These variables represent the logical result of the command (`True`/`False`) and result (a "return value", something computed), much like with *Expressions*.

For reference of current commands, see *Command Reference*.

If you're missing some cool functionality, you can implement your own command runner and send us a pull request or include it in `files` shipped with your assistants. Command runners shipped with assistants must be loaded with *load_cmd command runner*. Each command must be a class with two classmethods:

```
from devassistant.command_runners import command_runners
from devassistant.logger import logger

# NOTE: Command runners included in DA itself are decorated with @register_command_runner
#  wrapper. If you're shipping your own commands runners with assistants, don't do this.
class MyCommandRunner(CommandRunner):
    @classmethod
    def matches(cls, c):
        return c.comm_type == 'mycomm'
```

```
    @classmethod
    def run(cls, c):
        input = c.input_res
        logger.info('MyCommandRunner was invoked: {ct}: {ci}'.format(ct=c.comm_type,
                                                                     ci=input))
        return (True, len(input))
```

This command runner will run all commands with command type `mycomm`. For example if your assistant contains:

```
run:
- load_cmd: *file_from_files_section
- $foo: $(echo "using DevAssistant")
- mycomm: You are $foo!
```

than DevAssistant will print out something like:

```
INFO: MyCommandRunner was invoked: mycomm: You are using DevAssistant!
```

When run, this command returns a tuple with *logical result* and *result*. This means you can assign the length of a string to a variable like this:

```
run:
- $thiswillbetrue, $length~:
  - mycomm: Some string.
```

(Also, `LAST_LRES` will be set to `True` and `LAST_RES` to length of the input string.)

Generally, the `matches` method should just decide (True/False) whether given command is runnable or not and the `run` method should actually run it. The `run` method should use `devassistant.logger.logger` object to log any messages and it can also raise any exception that's subclass of `devassistant.exceptions.ExecutionException`.

The `c` argument of both methods is a `devassistant.lang.Command` object. You can use various attributes of `Command`:

- `comm_type` - command type, e.g. `mycomm` (this will always be stripped of exec flag `~`).
- `comm` - raw command input. The input is raw in the sense that it is uninterpreted. It's literally the same as what's written in assistant yaml file.
- `input_res` and `input_log_res` - result and logical result of `comm`, i.e. interpreted input. This is what you usually want to use to examine what was passed to your command. See *Section Results* for rules on interpreting command input.
- `had_exec_flag` - `True` if the command type had exec flag, `False` otherwise.

Note: input only gets evaluated one time - at time of using input_log_res or input_res. This means, among other things, that if exec flag is used, the command runner still has to access input_log_res or input_res to actually execute the input.

1.2.2 Create Your Own Assistant

Create Assistant in Yaml DSL

Tutorial: Creating Your Own Assistant in Yaml DSL

So you want to create your own assistant? There is nothing easier... They say that in all tutorials, right?

This tutorial will guide you through the process of creating simple assistants of *different roles* - Creator, Tweak, Preparer, Extras.

This tutorial doesn't cover everything. Consult *Yaml DSL Reference* when you're missing something you really need to achieve. If you think that DevAssistant misses some functionality that would be useful, open a bug at https://www.github.com/devassistant/devassistant/issues or send us a pull request.

General Rules Some things are common for all assistant types:

- Each assistant is one Yaml file, that must contain exactly one mapping - the so-called assistant attributes:

```
fullname: My Assistant
description: This will be part of help for this assistant
...
```

- You have to place them in a proper place, see *DevAssistant Load Paths* and *Assistants Loading Mechanism*.
- Files (e.g. templates, scripts, *PingPong script files* etc.) used by assistant should be placed in the same load dir, e.g. if your assistant is placed at `~/.devassistant/assistants`, DevAssistant will look for files under `~/.devassistant/files`.
- As mentioned in *DevAssistant Load Paths*, there are three main load paths in standard DevAssistant installation, "system", "local" and "user". The "system" dir is used for assistants delivered by your distribution/packaging system and you shouldn't touch or add files in this path. The "local" path can be used by system admin to add system-wide assistants while not touching "system" path. Lastly, "user" path can be used by user to install assistants just for himself.
- When developing new assistants, that you e.g. put in a separate Git repo and want to work on it, commit, push, etc, it is best to utilize `DEVASSISTANT_PATH` bash environment variable, see *DevAssistant Load Paths* for more info.

Creating a Simple Creator The title says it all. In this section, we will create a "Creator" assistant, that means an assistant that will take care of kickstarting a new project. We will write an assistant that creates a project containing a simple Python script that uses `argh` Python module. Let's suppose that we're writing this assistant for an RPM based system like Fedora, CentOS or RHEL.

To start, we'll create a file hierarchy for our new assistant, say in `~/programming` and modify `DEVASSISTANT_PATH` accordingly. Luckily, there is an assistant that does all this - dap:

```
da pkg install dap
da create dap -n ~/programming/pyargh --crt
export DEVASSISTANT_PATH=~/programming/pyargh/
```

Running `da create dap` scaffolds everything that's needed to create a DAP package that can be distributed on DevAssistant Package Index, DAPI, see *Packaging and Distributing Your Assistant* for more information.

Since this assistant is a "creator", we need to put it somewhere under `~/programming/assistants/crt/`. Assistants can be organized in a hierarchical structure, so you could have e.g. `~/programming/pyargh/assistants/crt/python-scripts.yaml` as a superassistant and `~/programming/pyargh/assistants/crt/python-scripts/pyargs.yaml` as its sub-assistant, but for this example we'll keep things simple and put `pyargh.yaml` directly under `~/programming/pyargh/assistants/crt/`.

Note, that in pre-0.10.0 DevAssistant versions, it was recommended to hook such assistants in already existing hierarchies (e.g. using superassistants provided by someone else). Since 0.10.0, this is no longer recommended. The main reason for this is that we are introducing a simple upstream packaging and distribution format, as well as "DevAssistant package index" - a central repository of upstream assistant packages. See *Packaging and Distributing Your*

Assistant for more details. In this concept, each package can only have one superassistant (named as the whole package is named) in each `crt`, `twk`, `prep` and `extra` and can only place subassistants into hierarchies defined by these. Package names have to be unique in the DevAssistant Package Index.

Setting it Up So, let's start writing `~/programming/pyargh/assistants/crt/pyargh.yaml` by providing some initial metadata:

```
fullname: Argh Script Template
description: Create a template of simple script that uses argh library
project_type: [python]
```

If you now save the file and run `da create pyargh -h`, you'll see that your assistant was already recognized by DevAssistant, although it doesn't provide any functionality yet. (Including project type in your Creator assistant is not necessary, but it may bring some benefits - see *Project Types*.

Dependencies Now, we'll want to add a dependency on `python-argh` (which is how the package is called e.g. on Fedora). You can do this just by adding:

```
dependencies:
- rpm: [python-argh]
```

Now, if you save the file and actually try to run your assistant with `da create pyargh`, it will install `python-argh`! (Well, assuming it's not already installed, in which case it will do nothing.) This is really super-cool, but the assistant still doesn't do any project setup, so let's get on with it.

Files Since we want the script to always look the same, we will create a file that our assistant will copy into proper place. This file should be put into into `crt/pyargh` subdirectory the files directory (`~/programming/files/crt/pyargh`). The file will be called `arghscript.py` and will have this content:

```
#!/usr/bin/python2

from argh import *

def main():
    return 'Hello world'

dispatch_command(main)
```

We will need to refer to this file from our assistant, so let's open `argh.yaml` again and add a `files` section:

```
files:
  arghs: &arghs
    source: arghscript.py
```

DevAssistant will automatically search for this file in the correct directory, that is `~/programming/files/crt/pyargh`. If an assistant has more subassistants, e.g. `crt/pyargh/someassistant` and these assistants need to share some files, it is reasonable to place them into `~/programming/files/crt/pyargh` and refer to them with relative path like `../file.foo` from the subassistants. Note, that the two `arghs` in `arghs: &arghs` should be the same because of issue 74.

Run Finally, we will be adding a `run` section, which is the section that does all the hard work. A `run` section is a list of **commands**. Every command is in fact a Yaml mapping with exactly one key and value. The key determines **command type**, while value is the **command input**. For example, `cl` is a **command type** that says that given **input** should be run on commandline, `log_i` is a **command type** that lets us print the **input** (message in this case) for user, etc.

Let's start writing our `run` section:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
```

But wait! We don't know what the project should be called and where it should be placed... Before we finish the `run` section, we'll need to add some arguments to our assistant.

Oh Wait, Arguments! Creating any type of project typically requires some user input, at least name of the project to be created. To ask user for this sort of information, we can use DevAssistant arguments like this:

```
args:
  name:
    flags: [-n, --name]
    required: True
    help: 'Name of project to create'
```

This means that this assistant will have one argument called `name`. On commandline, it will expect `-n foo` or `--name foo` and since the argument is required, it will refuse to run without it.

You can now try running `da create pyargh -h` and you'll see that the argument is printed out in commandline help.

Since there are some common arguments that the standard installation of DevAssistant ships with so called "snippets", that contain (among other things) definitions of frequentyl used arguments. You can use name argument for Creator assistants like this:

```
args:
  name:
    use: common_args
```

See *Common Assistant Behaviour* for more information.

Run Again Now that we're able to obtain project name (let's assume that it's an arbitrary path to a directory where the argh script should be placed), we can continue. First, we will make sure that the directory doesn't already exist. If so, we need to exit, because we don't want to overwrite or break something:

```
run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
- if $(test -e "$name"):
  - log_e: '"$name" already exists, can't proceed.'
```

There are few things to note here:

- There is a simple `if` condition with a shell command. If the shell command returns a non-zero value, the condition will evaluate to false, else it will evaluate to true. So in this case, if something exists at path `"$name"`, the condition will evaluate to true.
- In any command, we can use value of the `name` argument by prefixing argument name with `$` (so `$name` or `${name}`).
- The `log_e` command type is used to print a message and then abort the assistant execution immediately.

Let's continue by creating the directory. Add this line to `run` section:

```
- cl: mkdir -p "$name"
```

You may be wondering what will happen, if DevAssistant doesn't have write permissions or more generally if the `mkdir` command just fails. In this case, DevAssistant will exit, printing the output of failed command for user.

Next, we want to copy our script into the directory. We want to name it the same as name of the directory itself. But what if directory is a path, not simple name? We have to find out the project name and remember it somehow:

```
- $proj_name~: $(basename "$name")
```

What just happened? We assigned output of command `basename "$name"` to a new variable `proj_name` that we can use from now on. Note the `~` at the end of `$proj_name~`. This is called **execution flag** and it says that the command input should be executed as an expression, not taken as a literal. See *Expressions* for detailed expressions reference and *Variables and Context* to find out more about variables.

Note: the execution flag makes DevAssistant execute the input as a so-called "execution section". The input can either be a string, evaluated as an expression, or a list of commands, evaluated as another "run" section.

So let's copy the script and make it executable:

```
- cl: cp *arghs ${name}/${proj_name}.py
- cl: chmod +x ${name}/${proj_name}.py
```

One more thing to note here: by using `*arghs`, we reference a file from the `files` section.

Now, we'll use a super-special command:

```
- dda_c: "$name"
```

What is `dda_c`? The first part, `dda` stands for "dot devassistant file", the second part, `_c`, says, that we want to create this file (there are more things that can be done with `.devassistant` file, see *.devassistant Commands*). The "command" part of this call just says where the file should be stored, which is directory `$name` in our case.

The `.devassistant` file serves for storing meta information about the project. Amongst other things, it stores information about which assistant was invoked. This information can later serve to prepare the environment (e.g. install `python-argh`) on another machine. Assuming that we commit the project to a git repository, one just needs to run `da prepare custom -u <repo_url>`, and DevAssistant will checkout the project from git and use information stored in `.devassistant` to reinstall dependencies. (There is more to this, you can for example add a custom `run` section to `.devassistant` file or add custom dependencies, but this is not covered by this tutorial (see *Project Metainfo: the .devassistant File*).

Note: There can be more dependencies sections and run sections in one assistant. To find out more about the rules of when they're used and how run sections can call each other, consult dependencies reference and run reference.

Something About Snippets Wait, did we say Git? Wouldn't it be nice if we could setup a Git repository inside the project directory and do an initial commit? These things are always the same, which is exactly the type of task that DevAssistant should do for you.

Previously, we've seen usage of argument from snippet. But what if you could use a part of `run` section from there? Well, you can. And you're lucky, since there is a snippet called `git.init_add_commit`, which does exactly what we need. This snippet can be found in the git DAP. During development, you can install `git` DAP using `da pkg install git`. For runtime, you'll need to add it as dependency to `meta.yaml` - see *meta.yaml explained* for more info on dependencies. We'll use the snippet like this:

```
- cl: cd "$name"
- use: git.init_add_commit.run
```

This calls section `run` from snippet `git_init_add_commit` in this place. Note, that all variables are "global" and the snippet will have access to them and will be able to change their values. However, variables defined in called snippet section will not propagate into current section.

Finished! It seems that everything is set. It's always nice to print a message that everything went well, so we'll do that and we're done:

```
- log_i: Project "$proj_name" has been created in "$name".
```

The Whole Assistant ... looks like this:

```
fullname: Argh Script Template
description: Create a template of simple script that uses argh library
project_type: [python]

dependencies:
- rpm: [python-argh]

files:
  arghs: &arghs
    source: arghscript.py

args:
  name:
    use: common_args

run:
- log_i: Hello, I'm Argh assistant and I will create an argh project for you.
- if $(test -e "$name"):
  - log_e: '"$name" already exists, cannot proceed.'
- cl: mkdir -p "$name"
- $proj_name~: $(basename "$name")
- cl: cp *arghs ${name}/${proj_name}.py
- cl: chmod +x *arghs ${name}/${proj_name}.py
- dda_c: "$name"
- cl: cd "$name"
- use: git_init_add_commit.run
- log_i: Project "$proj_name" has been created in "$name".
```

And can be run like this: `da create pyargh -n foo/bar`.

Creating a Tweak Assistant *This section assumes that you've read the previous tutorial and are therefore familiar with DevAssistant basics.* Tweak assistants are meant to work with existing projects. They usually try to look for `.devassistant` file of the project, but it is not necessary.

Tweak Assistant Specialties **The special behaviour of tweak assistants only applies if you use dda_r in pre_run section. This command reads .devassistant file from given directory and puts the read variables in global variable context, so they're available from all the following dependencies and run section.**

If tweak assistant reads `.devassistant` file in `pre_run` section, DevAssistant tries to search for more `dependencies` sections to use. If the project was previously created by `crt python django`, the engine will install dependencies from sections `dependencies_python_django`, `dependencies_python` and `dependencies`.

Also, the engine will try to run `run_python_django` section first, then it will try `run_python` and then `run` - note, that this will only run the first found section and then exit, unlike with dependencies, where all found sections are used.

– IN PROGRESS –

Yaml DSL Reference

Note: The Yaml DSL has changed significantly in 0.9.0 in backwards incompatible manner. This documentation is only for version 0.9.0 and later.

This is a reference manual to writing yaml assistants. Yaml assistants use a special DSL defined on this page. For real examples, have a look at assistants in our Github repo.

Why the hell another DSL? When we started creating DevAssistant and we were asking people who work in various languages whether they'd consider contributing assistants for those languages, we hit the "I'm not touching Python" barrier. Since we wanted to keep the assistants consistent (centralized logging, sharing common functionality, same backtraces, etc...), we created a new DSL. This DSL is very well suited for what it's supposed to do, but we understand that some people don't want to learn it for various reasons. That is why, in 0.10.0, we introduced a concept called *DevAssistant PingPong*. Using PingPong, you can write assistants in scripting languages, while still utilizing DevAssistant as a "library of functions".

Assistant Roles For list and description of assistant roles see *Assistant Roles*.

The role is implied by assistant location in one of the load path directories, as mentioned in *Assistants Loading Mechanism*.

All the rules mentioned in this document apply to all types of assistants, with exception of sections *Tweak Assistants*, *Preparer Assistants* and *Extras Assistants* that talk about specifics of Tweak, resp. Preparer, resp. Extras assistants.

Assistant Name Assistant name is a short name used on command line, e.g. `python`. Historically, it had to be the only top-level yaml mapping in the file, e.g.:

```
python:
  fullname: Python
  description: Some verbose description
```

Since DevAssistant 0.9.0, it is preferred to omit it and just provide the assistant attributes as the top level mapping:

```
fullname: Python
description: Some verbose description
```

Assistant name is derived from the filename by stripping the `.yaml` extension, e.g. assistant `python.yaml` file is named `python`.

Assistant Attributes Assistant attributes form the top level mapping in Yaml file:

```
fullname: Python

run:
- cl: mkdir -p $name
- log_i: I'm in $name
```

List of allowed attributes follows (all of them are optional, and have some sort of reasonable default, it's up to your consideration which of them to use):

`fullname` a verbose name that will be displayed to user (`Python Assistant`)

`description` a (verbose) description to show to user (`Bla bla create project bla bla`)

`dependencies` (and `dependencies_*`) specification of dependencies, see below Dependencies

`args` specification of arguments, see below Arguments

`files` specification of used files, see below Files

project_type type of the project, see *Project Types*

run (and run_*) specification of actual operations, see *Run Sections Reference*

pre_run and post_run specification of operations to carry out before/after running main `run` section, see below Assistants Invocation; follow the rules specified in *Run Sections Reference*

files_dir directory where to take files (templates, helper scripts, ...) from. Defaults to base directory from where this assistant is taken + `files`. E.g. if this assistant is `~/.devassistant/assistants/crt/path/and/more.yaml`, files will be taken from `~/.devassistant/files/crt/path/and/more` by default.

icon_path absolute or relative path to icon of this assistant (will be used by GUI). If not present, a default path will be used - this is derived from absolute assistant path by replacing `assistants` by `icons` and `.yaml` by `.svg` - e.g. for `~/.devassistant/assistants/crt/foo/bar.yaml`, the default icon path is `~/.devassistant/icons/crt/foo/bar.svg`

Assistants Invocation When you invoke DevAssistant with it will run following assistants sections in following order:

- `pre_run`
- `dependencies`
- `run` (possibly different section for Tweak Assistants)
- `post_run`

If any of the first three sections fails in any step, DevAssistant will immediately skip to `post_run` and the whole invocation will be considered as failed (will return non-zero code on command line and show "Failed" in GUI).

Dependencies Yaml assistants can express their dependencies in multiple sections.

- Packages from section `dependencies` are **always** installed.
- If there is a section named `dependencies_foo`, then dependencies from this section are installed **iff** `foo` argument is used (either via commandline or via gui). For example:

  ```
  $ da python --foo
  ```

- These rules differ for Tweak Assistants

Each section contains a list of mappings `dependency type: [list, of, deps]`. If you provide more mappings like this:

```
dependencies:
- rpm: [foo]
- rpm: ["@bar"]
```

they will be traversed and installed one by one. Supported dependency types:

rpm the dependency list can contain RPM packages or YUM groups (groups must begin with `@` and be quoted, e.g. `"@Group name"`)

use / call (these two do completely same, call is obsolete and will be removed in 0.9.0) installs dependencies from snippet/another dependency section of this assistant/dependency section of superassistant. For example:

```
dependencies:
- use: foo.dependencies
- use: foo.dependencies_bar # will install dependencies from snippet "foo", section "bar"
```

```
  - use: self.dependencies_baz # will install dependencies from section "dependencies_baz" of this
  - use: super.dependencies # will install dependencies from "dependencies" section of first super
```

if, else conditional dependency installation. For more info on conditions see *Run Sections Reference*. A very simple example:

```
dependencies:
- if $foo:
  - rpm: [bar]
- else:
  - rpm: [spam]
```

Full example:

```
dependencies: - rpm: [foo, "@bar"]

dependencies_spam:
- rpm: [beans, eggs]
- if $with_spam:
  - use: spam.spamspam
- rpm: ["ham${more_ham}"]
```

Sometimes your dependencies may get terribly complex - they depend on many parameters, you need to use them dynamically during `run`, etc. In these cases, consider using *Dependencies Command* in `run` section.

Arguments Arguments are used for specifying commandline arguments or GUI inputs. Every assistant can have zero to multiple arguments.

The `args` section of each yaml assistant is a mapping of arguments to their attributes:

```
args:
  name:
    flags:
    - -n
    - --name
  help: Name of the project to create.
```

Available argument attributes:

flags specifies commandline flags to use for this argument. The longer flag (without the `--`, e.g. `name` from `--name`) will hold the specified commandline/gui value during `run` section, e.g. will be accessible as `$name`.

help a help string

required one of `{true,false}` - is this argument required?

nargs how many parameters this argument accepts, one of `{0, ?,*,+}` (e.g. {0, 0 or 1, 0 or more, 1 or more})

default a default value (this will cause the default value to be set even if the parameter wasn't used by user)

action one of `{store_true, [default_iff_used, value]}` - the `store_true` value will create a switch from the argument, so it won't accept any parameters; the `[default_iff_used, value]` will cause the argument to be set to default value `value` **iff** it was used without parameters (if it wasn't used, it won't be defined at all)

metavar a name of variable to show in help on command line, e.g. with `metavar: META`, you'll get a help line `--some-arg META <help>`.

use name of the snippet to load this argument from; any other specified attributes will override those from the snippet By convention, some arguments should be common to all or most of the assistants. See *Common Assistant Behaviour*

preserved if set, the value of this argument will be saved and will reappear in the next launch of devassistant GUI. The attribute string is a key under which the argument value will be stored. The key should be of the form "scope.argname" so that you can either share the value across more assistants or avoid collisions if any other assistant uses an argument with same name but different meaning. The argument values are stored in "~/.devassistant/.config". It is ignored in command-line interface.

Gui Hints GUI needs to work with arguments dynamically, choose proper widgets and offer sensible default values to user. These are not always automatically retrieveable from arguments that suffice for commandline. For example, GUI cannot meaningfully prefill argument that says it "defaults to current working directory". Also, it cannot tell whether to choose a widget for path (with the "Browse ..." button) or just a plain text field.

Because of that, each argument can have `gui_hints` attribute. This can specify that this argument is of certain type (path/str/bool) and has a certain default. If not specified in `gui_hints`, the default is taken from the argument itself, if not even there, a sensible "empty" default value is used (home directory/empty string/false). For example:

```
args:
  path:
    flags:
    - [-p, --path]
    gui_hints:
      type: path
      default: $(pwd)/foo
```

If you want your assistant to work properly with GUI, it is good to use `gui_hints` (currently, it only makes sense to use it for `path` attributes, as `str` and `bool` get proper widgets and default values automatically).

Files This section serves as a list of aliases of files stored in one of the `files` dirs of DevAssistant. E.g. if your assistant is `assistants/crt/foo/bar.yaml`, then files are taken relative to `files/crt/foo/bar/` directory. So if you have a file `files/crt/foo/bar/spam.foo`, you can use:

```
files:
  spam: &spam
    source: spam.foo
```

This will allow you to reference the `spam.foo` file in `run` section as `*spam` without having to know where exactly it is located in your installation of DevAssistant. Note, that the Yaml anchor name should be the same as mapping name, e.g. the two `spam` in `spam: &spam` should match. This is because of issue 74, that can't really be reasonably fixed.

Run Reference for run sections has a separate page: *Run Sections Reference*.

Creator Assistants Creator assistants are assistants that *create* something, be it a source file, a configuration file template or a whole new project. They must be placed under `assistants/crt` subdirectory or one of the load paths, as mentioned in *Assistants Loading Mechanism*.

They usually create `.devassistant` file (see *Project Metainfo: the .devassistant File*).

Tweak Assistants Tweak assistants are assistants that are supposed to work with already created project. They must be placed under `assistants/twk` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

There are few special things about tweak assistants:

- They usually utilize `dda_r` to read the whole `.devassistant` file (usually from directory specified by `path` variable or from current directory). *Since version 0.8.0, every tweak assistant has to do this on its own, be it in pre_run or run section*. This also allows you to work non-devassistant projects - just don't use `dda_r`.

The special rules below *only apply if you use dda_t in pre_run section*.

- They use dependency sections according to the normal rules + they use *all* the sections that are named according to `project_type` loaded from `.devassistant`, e.g. if `project_type` is `[foo, bar]`, dependency sections `dependencies`, `dependencies_foo` and `dependencies_foo_bar` will be used as well as any sections that would get installed according to specified parameters. The rationale behind this is, that if you have e.g. `eclipse` tweak assistant that should work for both `python django` and `python flask` projects, chance is that they have some common dependencies, e.g. `eclipse-pydev`. So you can just place these common dependencies in `dependencies_python` and you're done (you can possibly place special per-framework dependencies into e.g. `dependencies_python_django`).
- By default, they don't use `run` section. Assuming that `project_type` is `[foo, bar]`, they first try to find `run_foo_bar`, then `run_foo` and then just `run`. The first found is used. If you however use cli/gui parameter `spam` and section `run_spam` is present, then this is run instead.

Preparer Assistants Preparer assistants are assistants that are supposed to checkout sources of upstream projects and set up environment for them (possibly utilizing their `.devassistant` file, if they have one). Preparers must be placed under `assistants/prep` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*.

Preparer assistants commonly utilize the `dda_dependencies` and `dda_run` commands in `run` section.

Extras Assistants Extras assistants are supposed to carry out arbitrary task that are not related to a specific project. They must be placed under `assistants/extra` subdirectory of one of the load paths, as mentioned in *Assistants Loading Mechanism*. Otherwise, there is nothing special about extras assistants in terms of execution by DevAssistant.

Run Sections Reference

Run sections are the essence of DevAssistant. They are responsible for performing all the tasks and actions to set up the environment and the project itself. For Creator and Preparer assistants, the section named `run` is always invoked, *Tweak Assistants* may invoke different sections based on metadata in a `.devassistant` file.

Note, that `pre_run` and `post_run` follow the same rules as `run` sections. See *Assistants Invocation* to find out how and when these sections are invoked.

Every section is a sequence of various **commands**, mostly invocations of commandline. Each command is a mapping of **command type** to **command input**:

```
run:
- command_runner: command_input
- command_runner_2: another_command_input
```

Note, that **section** is a general term used for any sequence of commands. Sections can have subsections (e.g. in conditions or loops), assuming they follow some rules (see below).

Introduction to Commands and Variables The list of all supported commands can be found at *Command Reference*, we only document the basic usage of the most important commands here. Note, that when you use variables (e.g. `$variable`) in command input, they get substituted for their values (undefined variables will remain unchanged).

- command line invocation:

```
- cl: mkdir -p $spam
```

This will invoke a subshell and create a directory named `$spam`. If the command returns non-zero return code, DevAssistant will fail.

- logging:

```
- log_i: Directory $spam created.
```

This command will log the given message at `INFO` level - either to terminal or GUI. You can use similar commands to log at different log levels: `log_d` for `DEBUG`, `log_w` for `WARNING`, `log_e` for `ERROR` and `log_c` for `CRITICAL`. By default, messages of level `INFO` and higher are logged. Log messages with levels `ERROR` and `CRITICAL` emit the message and then *raise an exception*.

- conditions:

```
- if not $foo and $(ls /spam/spam/spam):
  - log_i: This gets executed if the condition is satisfied.
- else:
  - log_i: Else this section gets executed.
```

Conditions work as you'd expect in any programming language - `if` subsection gets executed if the condition evaluates to true, otherwise `else` subsection gets executed. The condition itself is an **expression**, see *Expressions* for detailed reference of expressions.

- loops:

```
- for $i word_in $(ls):
  - log_i: Found file $i.
```

Loops probably also work as you'd expect - they've got the control variable and an iterable. Loop iterators are **expressions**, see *Expressions*. Note, that you can use two forms of for loop. If you use `word_in`, DevAssistant will split the given expression on whitespace and then iterate over that, while if you use `in`, DevAssistant will iterate over single characters of the string.

- variable assignment:

```
- $foo: "Some literal with value of "foo" variable: $foo"
```

This shows how to assign a literal value to a variable. It is also possible to assign the result of another command to a variable, see Section Results for how to use the execution flag.

Remember to check *Command Reference* for a comprehensive description of all commands.

Literal Sections vs. Execution Sections DevAssistant distinguishes two different section types: **input sections** and **execution sections**. Some sections are inherently execution sections:

- all `run` sections of assistants
- `if`, `else` subsections
- `for` subsections

Generally, execution sections can be either:

- *expression* (e.g. a Yaml string that gets interpreted as an expression)

or

- section (sequence of **commands**)

Literal section can be any valid Yaml structure - string, list or mapping.

Section Results Similarly to *expressions*, sections return *logical result* and *result*:

- literal section
  - *result* is a string/list/mapping with variables substituted for their values
  - *logical result* is False if the structure is empty (empty string, list or mapping), True otherwise
- execution sections
  - *result* is the result of last command of given section
  - *logical result* is the logical result of last command of given section

Some examples follow:

```
run:
# now we're inherently in an execution section
- if $(ls /foo):
  # now we're also in an execution section, e.g. the below sequence is executed
  - foo:
      # the input passed to "foo" command runner is inherently a literal input, e.g. not executed
      # this means foo command runner will get a mapping with two key-value pairs as input, e.g.:
      # {'some': 'string value', 'with': [ ... ]}
      some: string value
      with: [$list, $of, $substituted, $variables]
- $var: this string gets assigned to "var" with $substituted $variables
```

If you need to assign the result of an expression or execution section to a variable or pass it to a command runner, you need to use the **execution flag**: ~:

```
run:
- $foo~: ($this or $gets) and $executed_as_expression
- foo~:
  # input of "foo" command runner will be result of the below execution section
  - command_runner: literal_section
  - command_runner_2~:
    # similarly, input of command_runner_2 will be result of the below execution section
    - cr: ci
    - cr2: ci2
```

Note, that a string starting with the execution flag is also executed as an expression. If you want to create a literal that starts with ~, just use the escape value for it (~~):

```
run:
- $foo: ~$(ls) and $bar
- $bar: ~~/some_dir_in_users_home
- log_i: The tilde character (~) only needs to be escaped when starting a string.
```

Each command specifies its return value in a different way, see *Command Reference*.

Exceptions If an unexpected error happens in a command runner, then this command runner *raises exception*. This means that execution of the current section is immediately terminated - in fact, the whole assistant run is terminated at that moment. In terminology terms, this is called *raising exception* even for a run section. As of version 0.10.0, there is no way to recover from these errors, but it may be added in future versions.

For command line execution of DevAssistant, raising exception means ending DevAssistant with non-zero return code immediately. In GUI, this means ending the execution of an assistant, but keeping the GUI running.

Variables and Context The set of all variables existing during an assistant `run` section is referred to as *global context* or just *context* (it is implemented as dictionary, Python's associative array type). This means, that it is in fact mapping of variable names to their values.

Initially, the context is populated with values of arguments from the commandline/gui and some other useful values, see *Global Variables* below. You can of course define (and assign to) your own variables or change the values of current ones - see *Variable Assignment*. Names of some of the preset variables start and end with double underscores. You shouldn't modify these, as they can be used internally by DevAssistant.

Additionally, after each command, variables `$LAST_RES` and `$LAST_LRES` are populated with the result of the last command (these are also the return values of the command) - see *Command Reference*.

The variable scope works as follows:

- When invoking a different `run` section (from the current assistant or snippet), the variables get passed by value (e.g. they don't get modified for the remainder of this scope).
- Variables defined in subsections (`if`, `else`, `for`) continue to be available until the end of the current `run` section.

All variables are global in the sense that if you call a snippet or another section, it can see all the arguments that are defined.

Quoting When using variables that contain user input, they should always be quoted in the places where they are used for bash execution. That includes `cl*` commands, conditions that use bash return values and variable assignment that uses bash.

Global Variables In all assistants, a few useful global variables are available. These include:

- `$__system_name__` - name of the system, e.g. "linux"
- `$__system_version__` - version of the system, e.g. "3.13.3-201.fc20.x86_64"
- `$__distro_name__` - name of Linux distro, e.g. "fedora"
- `$__distro_version__` - version of Linux distro, e.g. "20"
- `$__env__` - mapping of environment variables that get passed to subprocess shell

Note: if any of this information is not available, the corresponding variable will be empty. Also note, that you can rely on all the variables having lowercase content.

Expressions Expressions are used in assignments, conditions and as loop "iterables". Every expression has a *logical result* (meaning success - `True` or failure - `False`) and *result* (meaning output). *Logical result* is used in conditions and variable assignments, *result* is used in variable assignments and loops. Note: when assigned to a variable, the *logical result* of an expression can be used in conditions as expected; the *result* is either `True` or `False`.

Syntax and semantics:

- `$foo`
  - if `$foo` is defined:
    - *logical result*: `True` *iff* value is not empty and it is not `False`
    - *result*: value of `$foo`
  - otherwise:
    - *logical result*: `False`
    - *result*: empty string

  - *note*: boolean values (e.g. those acquired by argument with `action: store_true`) always have an empty string as a *result* and their value as *logical result*

- `$(commandline command)` (yes, that is a command invocation that looks like running command in a subshell)
  - if `commandline command` has return value 0:
    * *logical result*: `True`
  - otherwise:
    * *logical result*: `False`
  - regardless of *logical result*, *result* always contains both stdout and stderr lines in the order they were printed by `commandline command`
  - *note*: Due to the way the expression parser works, DevAssistant may sometimes add spaces around special characters between `$(` and `)`. This is a known issue, but we don't have any systematic solution right now. The problem can be worked around by putting quotes (single or double) around the whole commandline invocation, e.g. you can use `$("echo +-")`. See *issue 271 <https://github.com/devassistant/devassistant/issues/271>*.
- `as_root $(commandline command)` runs `commandline command` as superuser; DevAssistant may achieve this differently on different platforms, so the actual way how this is done is considered to be an implementation detail
- `defined $foo` - works exactly as `$foo`, but has *logical result* `True` even if the value is empty or `False`
- `not $foo` negates the *logical result* of an expression, while leaving *result* intact
- `$foo and $bar`
  - *logical result* is the logical conjunction of the two arguments
  - *result* is an empty string if at least one of the arguments is empty, or the latter argument
- `$foo or $bar`
  - *logical result* is the logical disjunction of the two arguments
  - *result* is the first non-empty argument or an empty string
- `literals - "foo", 'foo'`
  - *logical result* `True` for non-empty strings, `False` otherwise
  - *result* is the string itself, sans quotes
  - *Note: If you use an expression that is formed by just a literal, e.g.* `"foo"` *, then DevAssistant will fail, since Yaml parser will strip these. Therefore you have to use* `'"foo"'` *.*
- `$foo in $bar`
  - *logical result* is `True` if the result of the second argument contains the result of the second argument (e.g. "inus" in "Linus Torvalds") and `False` otherwise
  - *result* is always the first agument

All these can be chained together, so, for instance, `"1.8.1.4" in $(git --version) and defined $git` is also a valid expression

Snippets

Snippets are the DevAssistant's way of sharing common pieces of assistant code. For example, if you have two assistants that need to log identical messages, you want the messages to be in one place, so that you don't need to change them twice when a change is needed.

Example Let's assume we have two assistants like this:

```
### assistants/crt/assistant1.yaml
...
run:
- do: some stuff
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
- do: more stuff

### assistants/crt/assistant2.yaml
...
run:
- do: some slightly different stuff
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
- do: more slightly different stuff
```

So we have two assistants that have three lines of identical code in them - that breaks a widely known programmer best practice: Don't do it twice, write a function for it. In DevAssistant terms, we'll write a run section and place it in a snippet:

```
### snippets/mysnip.yaml
run:
- log_i: Creating cool project $name ...
- log_i: Still creating ...
- log_i: I suggest you go have a coffee ...
```

Then we'll change the two assistants like this (we'll utilize *"use" command runner*):

```
### assistants/crt/assistant1.yaml
...
run:
- do: some stuff
- use: mysnip.run
- do: more stuff

### assistants/crt/assistant2.yaml
...
run:
- do: some slightly different stuff
- use: mysnip.run
- do: more slightly different stuff
```

How Snippets Work This section summarizes important notes about how snippets are formed and how they work.

Syntax and Sections Snippets are very much like assistants. They can (but don't have to) have *args*, *dependencies** and *run** sections - structured in the same manner as in assistants. A snippet can contain any combination of the above sections (even empty file is a valid snippet).

Variables When a snippet section is called (this applies to both *dependencies** and *run**, it gets a copy of all arguments of its caller - e.g. it can use the variables, it can assign to them, but they'll be unchanged in the calling section after the snippet finishes.

Using Snippets and Return Value As noted above, snippets can hold 3 types of content (*args*, *dependencies** sections and *run** sections), each of which can be used in assistants:

```
### snippets/mysnip.yaml

args:
  foo:
    flags: [-f, --foo]
    help: Foo is foo
    required: True

dependencies:
- rpm: [python3]

run:
- log_i: Spam spam spam

### assistants/crt/assistant1.yaml

args:
  foo:
    use: mysnip

dependencies:
- use: mysnip.dependencies

run:
- do: stuff
- use: mysnip.run
```

Return values (RES and LRES) of snippet are determined by the *use command runner* - RES and LRES of last command of the snippet section.

Using DevAssistant Yaml DSL is the first option to create assistants. The DSL is fairly simple and understandable and is very good at what it does. However, it's not well suited for very complex computations (which you usually don't need to do during project setup).

If, for some reason, you need to execute complex algorithms in assistants (or you just don't want to learn the DSL), you can consider using the *PingPong approach*, which basically lets you write assistants in popular scripting languages.

Create Assistant Using Scripting Language (a.k.a DevAssistant PingPong)

How It Works/PingPong Protocol Reference

The DevAssistant PingPong protocol is a simple protocol that DevAssistant (Server) and PingPong script (Client) use to communicate through a pipe. It's designed to be as simple and portable as possible.

The overall idea of PingPong is:

- Server invokes Client script as a subprocess and attaches to its stdin/stdout/stderr, creating a pipe.
- Client waits for Server to initiate the communication.
- Server sends the first message, initiating the communication.
- Server and Client communicate through the pipe.
- At one point, the Client is done and the subprocess exits. Server gathers its output data and acts up on them in some way.

Right now, only Python implementation of the protocol is available. In future, we'll be aiming to implement the Client side (used in PingPong scripts) in other dynamic languages (or you can do it yourself using the reference below and let us know!)

Why PingPong? The "PingPong" name comes from the similarity to table tennis. There are two players, Server (DevAssistant) and Client (PingPong script). The Server serves (sends the first message), Client receives it and responds to server, Server receives the message and responds again to Client, ...

How Does PingPong Integrate With DSL? In terms of integration with DevAssistant Yaml DSL, PingPong is just another *command runner* that computes something and then returns a result. This means that you can mix it up arbitrarily with other DSL commands or even run several PingPong scripts in one assistant.

Reference This part describes DevAssistant PingPong Protocol (**DAPP**) version **2**. There is a reference Python implementation, called `dapp`, which you can examine into detail. Note that the reference implementation implements both Server and Client side. If you're considering implementing DAPP in another scripting language, you'll only need to implement Client side.

Errors Throughout this reference, there are certain situations marked as *"being an error"*. These situations usually mean that a (fatal) error was encountered in message format. The side getting the error should terminate immediately, possibly running cleanup code first.

Message Format These points are general rules that apply to all messages, both sent from Server to Client (S->C) and Client to Server (C->S).

- Currently, sending random binary data is not supported, everything has to be valid UTF-8 encoded string. Not being able to decode is an error.
- Each message starts with string `START` and ends with string `STOP`. These have to be on separate lines.
- Any non-empty line between previous `STOP` and following `START` is an error.
- The lines between `START` and `STOP` must a valid Yaml mapping, otherwise it is an error.
- Every message has to contain `msg_number`, `msg_type` and `dapp_protocol_version`.
- `dapp_protocol_version` must be an integer specifying the DAPP protocol version. Other side using a different protocol version is an error.
- `msg_number` must be a unique integer identifying the message during a PingPong script run. Sequence of message numbers must be increasing; both sides use the same sequence (e.g. Server sends message `1`, client then has to send message with number no lower than `2`, then Server has to send a message with number no lower than the number of message sent by client etc). This rule has one exception, confirmation messages (`msg_type` is `msg_received`) have the same number as the message that they're confirming. More on the confirmation messages below.

- If `msg_type` is different than `msg_received`, message must contain `ctxt`. Valid message types are listed below.
- `ctxt` has to be a Yaml DSL context (e.g. mapping of variable names to their values). In every message (except confirmation message), the whole context has to be passed and the receiving communication side must update its copy of the context.

Message Types and Content

- Both Client and Server send `msg_received` messages to confirm messages received from the other communicating side.
- Server sends these messages:
  - `run` - This message must always be the first message in the whole communication, Server sends it to tell Client to start and pass the initial context. This message shouldn't contain any special data.
  - `command_result` - Reports result of a command that Client previously invoked. Must contain `lres` and `res` values, these two represent results of the command (see *Command Reference* for details).
  - `command_exception` - Sent if the command called by Client raised an exception. Must contain `exception` value, which is a string representation of the exception.
  - `no_such_command` - Sent if Server (DevAssistant) doesn't know how to execute the sent command. Doesn't contain any extra data.
- Client sends these messages:
  - `call_command` - Client calls a command. Must contain `command_type` and `command_input`, as specified in *Command Reference*.
  - `finished` - Client ended successfully. Must contain `lres` and `res` values. These must be the same types as return values of DevAssistant commands (again, see *Command Reference*).
  - `failed` - Client failed. Must contain `fail_desc` with the description of the failure.

If `run` from Server to Client isn't the first message and `finished` or `failed` isn't the last message from Client to Server, it is an error.

Example Communication To illustrate better how the protocol works, here is a simple example of valid message sequence. We're assuming that the Server has already started the Client and Client is now waiting to for Server to initiate the communication.

Note that all the communication is shown as an already decoded Unicode, but in fact it's sent as UTF-8 through the pipe.

Server initiates communication by sending `run` message:

```
START
dapp_protocol_version: 2
msg_type: run
msg_number: 1
ctxt:
  name: user_input_name
  some_list_variable: [foo, bar, baz]
STOP
```

Client confirms that it got the message:

```
START
dapp_protocol_version: 2
msg_type: msg_received
msg_number: 1
STOP
```

And imediatelly after that it starts to actually do something. At certain points, it needs to call back to Server (DevAssistant) to carry out some tasks implemented in DevAssistant itself. Note, that while computing, the Client process has done some modifications to the context:

```
START
dapp_protocol_version: 2
msg_type: call_command
msg_number: 2
ctxt:
  name: user_input_name
  some_dict_variable: {foo: a, bar: b, baz: c}
command_type: log_i
command_input: This will get logged by DevAssistant to either GUI or console.
STOP
```

The Server (DevAssistant) first confirms receiving the message by sending `msg_received`:

```
START
dapp_protocol_version: 2
msg_type: msg_received
msg_number: 2
STOP
```

Then the server actually runs the command and sends a message with result to Client:

```
START
dapp_protocol_version: 2
msg_type: command_result
msg_number: 3
ctxt:
  name: user_input_name
  some_dict_variable: {foo: a, bar: b, baz: c}
lres: True
res: This will get logged by DevAssistant to either GUI or console.
STOP
```

(Note that for the `log_i` command, the `res` result is actually equal to the input; this is usually not the case, of course).

Again, Client confirms receiving the message:

```
START
dapp_protocol_version: 2
msg_type: msg_received
msg_number: 3
STOP
```

And then Client continues to compute. Since this is a simple example, the Client doesn't call any more commands, but it could call as many as it'd like. The client is now finished and prepared to exit, so it sends a `finished` message:

```
START
dapp_protocol_version: 2
msg_type: command_result
msg_number: 4
ctxt:
  name: user_input_name
```

```
  some_dict_variable: {foo: a, bar: b, baz: c}
  another_variable: some_var
lres: True
res: 42
STOP
```

Server sends one last confirmation message to Client:

```
START
dapp_protocol_version: 2
msg_type: msg_received
msg_number: 4
STOP
```

And everything is done. The Client can safely exit and Server can do anything it wishes with the result.

Tutorial: Assistants Utilizing PingPong

Regardless of which language you want to choose for implementing the PingPong script, you should read this section. It provides general information about specifying metadata, dependencies, arguments and file placement for the PingPong scripts.

DevAssistant distinguishes four different *assistant roles* - Creator, Tweak, Preparer, Extras. From the point of view of this tutorial, the roles only differ in file placement and where they'll be presented to user on command line/in GUI. Therefore we choose to create a simple Creator. We'll be implementing an assistant, that creates a simple reStructuredText document.

Note on terminology: The PingPong script is not the assistant. Even if you're using the PingPong approach, the assistant is still a Yaml file (very simple in this case).

General Rules Some things are common for all assistant types:

- Each assistant is one Yaml file, that must contain exactly one mapping - the so-called assistant attributes:

  ```
  fullname: My Assistant
  description: This will be part of help for this assistant
  ...
  ```

- You have to place them in a proper place, see *DevAssistant Load Paths* and *Assistants Loading Mechanism*.
- Files (e.g. templates, scripts, *PingPong script files* etc.) used by assistant should be placed in the same load dir, e.g. if your assistant is placed at `~/.devassistant/assistants`, DevAssistant will look for files under `~/.devassistant/files`.
- As mentioned in *DevAssistant Load Paths*, there are three main load paths in standard DevAssistant installation, "system", "local" and "user". The "system" dir is used for assistants delivered by your distribution/packaging system and you shouldn't touch or add files in this path. The "local" path can be used by system admin to add system-wide assistants while not touching "system" path. Lastly, "user" path can be used by user to install assistants just for himself.
- When developing new assistants, that you e.g. put in a separate Git repo and want to work on it, commit, push, etc, it is best to utilize `DEVASSISTANT_PATH` bash environment variable, see *DevAssistant Load Paths* for more info.

Getting Set Up To get started, we'll create a file hierarchy for our new assistant, say in `~/programming`. We'll also modify `DEVASSISTANT_PATH` so that DevAssistant can see this assistant in directory outside of standard load paths. Luckily, there is assistant that does all this - dap:

```
da pkg install dap
da create dap -n ~/programming/rstcreate --crt
export DEVASSISTANT_PATH=~/programming/rstcreate/
```

Running `da create dap` scaffolds everything that's needed to create a DAP package that can be distributed on DevAssistant Package Index, DAPI, see *Packaging and Distributing Your Assistant* for more information.

Since this assistant is a Creator, we need to put it somewhere under `assistants/crt` directory. The related files (if any), including the PingPong script have to go under `files/crt/rstcreate` (assuming, of course, we name the assistant `rstcreate.yaml`). More details on assistants file locations and subassistants can be found in the *tutorial for the Yaml DSL*.

Now go to one of the language-specific tutorials to see how to actually create a simple assistant and the PingPong script.

Tutorial: Python PingPong Script

This tutorial explains how to write a Python assistant using PingPong protocol. You should start by setting up the general things explained in *general tutorial*.

Creating the Yaml Assistant Since one of the points of PingPong is to avoid as much of the Yaml DSL as possible, this will be very short (and self-explanatory, too!). This is what you should put in `~/programming/rstcreate/assistants/crt/rstcreate.yaml`:

```
fullname: RST Document
description: Create a simple reStructuredText document.

dependencies:
- rpm: [python3, python3-dapp]

args:
  title:
    flags: [-t, --title]
    help: Title of the reStructuredText document.
    required: True

files:
  ppscript: &ppscript
    source: ppscript.py

run:
- pingpong: python3 *ppscript
```

This is pretty much all you'll need to write in the Yaml DSL everytime you'll be writing assistants based on PingPong. A brief explanation follows (more detailed explanation of the DSL can be found at *Tutorial: Creating Your Own Assistant in Yaml DSL*):

- `fullname` and `description` are "nice" attributes to show to users.
- `dependencies` list packages that DevAssistant is supposed to install prior to invoking the PingPong script; you can add any dependencies that your PingPong script needs here
- `args` are a Yaml mapping of arguments that the assistant will accept from user (be it on commandline or in GUI).
- `files` is a Yaml mapping of files; each file must a have a unique name (`ppscript`), should be referenced to by Yaml anchor (`&ppscript`; shouldn't be different from `ppscript` because of issue 74) and has to have `source` argument that specifies filename. (Will be searched for in appropriate `files` subdirectory.

- `run` just runs the PingPong script the way it's supposed to be run (the `python3 *ppscript`) is exactly what will get executed to execute the PingPong subprocess (of course after substituting `*ppscript` with expanded path to the actual script from `files`).

Creating the PingPong Script We'll write the PingPong script in Python 3, using the dapp library. Note, that this tutorial uses version 0.3.0 of dapp; consult dapp documentation if your version is different (you can find a detailed documentation of this library at its Github project page).

This is the content of the `~/programming/rstcreate/files/crt/rstcreate/ppscript.py` file (see comments below for explanation):

```
#!/usr/bin/python3
import os

import dapp

class MyScript(dapp.DAPPClient):
    def run(self, ctxt):
        # call a DA command that replaces funny characters by underscores,
        #  so that we can use title as a filename
        _, normalized = self.call_command(ctxt, 'normalize', ctxt['title'])
        filename = normalized.lower() + '.rst'

        # if file already exists, just fail
        if os.path.exists(filename):
            self.send_msg_failed(ctxt,
                'File "{0}" already exists, cannot continue!'.format(filename))

        self.call_command(ctxt, 'log_i', 'Creating file {0} ...'.format(filename))
        with open(filename, 'w') as f:
            # Issue a debug message that will show if DA is run with --debug
            self.call_command(ctxt, 'log_d', 'Writing to file {0}'.format(filename))
            f.write(ctxt['title'].capitalize())
            f.write('\n')
            f.write('=' * len(ctxt['title']))

        # inform user that everything went fine and return
        self.call_command(ctxt, 'log_i', 'File {0} was created.'.format(filename))
        return (True, filename)

if __name__ == '__main__':
    MyScript().pingpong()
```

- The PingPong script mustn't write anything to `stdout` or `stderr`. If you need to tell something to user, use `log_i` command (`log_w` for warnings and `log_d` for debug output).
- The whole PingPong script is just a Python 3 script that imports `dapp` library, subclasses the `dapp.DAPPClient` class and runs `pingpong()` method when script is run (*note: the implemented class has to implement run() method, but pingpong() has to be called!*).
- The `run` method takes `ctxt` as an argument, which is the Yaml DSL context. In short, it is a dictionary mapping DSL variables to their values. This context has to be passed as the first argument to all functions that interact with DevAssistant. Note, that all changes that you do to `ctxt` are permanent and will reflect in any subsequent Yaml DSL commands following the PingPong script invocation. See *Variables and Context* for more details on how context and variables work.
- You can run *DevAssistant commands* by calling `self.call_command` method. It takes three parameters: Yaml DSL context, *command type* and *command input* (consult *command_ref* for details on command types

and their input). This function returns 2-tuple, *logical result* (boolean) and *result* (type depends on command) (again, consult *command_ref*).

- You can pass arbitrary dictionaries (== crafted to make commands see a different context) to `call_command()` to achieve desired results. Doing this does *not* alter the Yaml DSL context in any way, the changes will be limited to the dictionary you pass.
- Similarly, the called commands can change the context that you pass to them as argument (usually they don't do this; if they do, they usually just add variables, not remove/change).
- The `run()` method has to return a 2-tuple, a *logical result* and *result*. This is exactly the same as what any DevAssistant command returns (since `pingpong` is in fact just a Yaml command). You can choose what you want to return as *result* as you wish - in this case, we return the name of the file created.

Wrap-up That is it. Now you can run the assistant with:

```
da create rstcreate -t "My Article"
```

And that's it. Enjoy!

(*Why PingPong?*)

The PingPong approach is the second approach you can take to write complex assistant functionality. It utilizes a small subset of the *Yaml DSL* for describing metadata, dependencies and assistant arguments.

The actual execution part is written in one of the supported *Supported Languages*. For each of these languages, there is a binding library available, that allows the PingPong script to make callbacks to DevAssistant. Hence you can write assistants in a scripting language while still utilizing DevAssistant functionality.

The general part about metadata, dependencies and arguments is described at *Tutorial: Assistants Utilizing PingPong* and you should read it regardless of the language you choose to implement the script. Then you should choose a language-specific tutorial to see how to write the actual PingPong script.

Supported Languages

Currently, only Python PingPong client library has been implemented. It works with Python 2.6, 2.7 and > 3.3. You can get it at *https://pypi.python.org/pypi/dapp*, bug reports/feature requests are welcome at *https://github.com/devassistant/dapp/issues*. This library is maintained by developers of DevAssistant.

Note on terminology: The PingPong script is not the assistant. Even if you're using the PingPong approach, the assistant is still a Yaml file (very simple in this case).

Common Assistant Behaviour

Common Parameters of Assistants and Their Meanings

`-e` Create Eclipse project, optional. Should create `.project` (or any other appropriate file) and register project to Eclipse workspace (`~/workspace` by default, or the given path if any).

`-g` Register project on GitHub (uses current user name by default, or given name if any).

`-n` Name of the project to create, mandatory. Should also be able to accept full or relative path.

`-p` Path to existing project supplied to tweak assistants (optional, defaults to `.`).

To include these parameters in your assistant with common help strings etc., include them from `common_args.yaml` (`-n`, `-g`) or `eclipse.yaml` (`-e`) snippet:

```
args:
  name:
    snippet: common_args
```

Other Conventions

When creating snippets/Python commands, they should operate under the assumption that current working directory is the project directory (not one dir up or anywhere else). It is the duty of assistant to switch to that directory. The benefit of this approach is that you just `cd` once in assistant and then call all the snippets/commands, otherwise you'd have to put 2x `cd` in every snippet/command.

Packaging and Distributing Your Assistant

Note: this functionality is under heavy development and is not fully implemented yet.

So now you know how to *create an assistant*. But what if you want to share your assistant with others?

For that you could send them all the files from your assistant and tell them where they belong. But that would be very unpleasant and that's why we've invented DAP. DAP is a format of extension for DevAssistant that contains custom assistants. It means DevAssistant Package.

A DAP is a tar.gz archive with `.dap` extension. The name of a DAP is always `<package_name>-<version>.dap` - i.e. `foo-0.0.1.dap`.

Directory structure of a DAP

The directory structure of a DAP copies the structure of `~/.devassistant` or `/usr/share/devassistant` folder. The only difference is, that it can only contain assistants, files and icons that that belongs to it's namespace.

Each DAP has an unique name (lat's say `foo`) and it can only contain assistants `foo` or `foo/*`. Therefore, the directory structure looks like this:

```
foo-0.0.1/
  meta.yaml
  assistants/
    {crt,twk,prep,extra}/
      foo.yaml
      foo/
  files/
    {crt,twk,prep,extra,snippets}/
      foo/
  snippets/
    foo.yaml
    foo/
  icons/
    {crt,twk,prep,extra,snippets}/
      foo.{png,svg}
      foo/
  doc/
      foo/
```

Note several things:

- Each of this is optional, i.e. you don't create `files` or `snippets` folder if you provide no files or snippets. Only mandatory thing is `meta.yaml` (see below).

- Everything goes to the particular folder, just like you've learned in the chapter about *creating assistants*. However, you can only add stuff named as your DAP (means either a folder or a file with a particular extension). If you have more levels of assistants, such as `crt/foo/bar/spam.yaml`, you have to include top-level assistants (in this case both `crt/foo.yaml` and `crt/foo/bar.yaml`). And you have to preserve the structure in other folders as well (i.e. no `icons/crt/foo/spam.svg` but `icons/crt/foo/bar/spam.svg`).
- The top level folder is named `<package_name>-<version>`.

meta.yaml explained There is an important file called `meta.yaml` in every DAP. It contains mandatory information about the DAP as well as additional optional metadata. Let's see an explained example:

```
package_name: foo # required
version: 0.0.1 # required
license: GPLv2 # required
authors: [Bohuslav Kabrda <bkabrda@mailserver.com>, ...] # required
homepage: https://github.com/bkabrda/assistant-foo # optional
summary: Some brief one line text # required
bugreports: <a single URL or email address> # optional
dependencies:
  # for now, dependencies are possible, but the version specifiers are ignored
  - bar
  - eggs >= 1.0
  - spam== 0.1     # as you can see, spaces are optional
  - ook   <    2.5 # and more can be added, however, don't use tabs
supported_platforms: [fedora, darwin] # optional
description: |
    Some not-so-brief optional text.
    It can be split to multiple lines.

    BTW you can use **Markdown**.
```

- **package name** can contain lowercase letters (ASCII only), numbers, underscore and dash (while it can only start and end with a letter or digit), it has to be unique, several names are reserved by DevAssitant itself (e.g. python, ruby)
- **version** follows this scheme: <num>[.<num>]*[dev|a|b], where 1.0.5 < 1.1dev < 1.1a < 1.1b < 1.1
- **license** is specified via license tag used in Fedora https://fedoraproject.org/wiki/Licensing:Main?rd=Licensing#Good_Licenses
- **authors** is a list of authors with their e-mail addresses (_at_ can be used instead of @)
- **homepage** is an URL to existing webpage that describes the DAP or contains the code (such as in example), only http(s) or ftp is allowed, no IP addresses
- **summary** and **description** are self-descriptive in the given example
- **bugreports** defines where the user should report bugs, it can be either an URL (issue tracker) or an e-mail address (mailing list or personal)
- **dependencies** specifies other DAPs this one needs to run - either non-versioned or versioned, optional; note, that versions are ignored for now, they'll start working in one of the future DevAssistant releases
- **supported_platforms** optionally lists all platforms (Linux distributions etc.), that this DAP is known to work on. When missing or empty, all platforms are considered supported. You can choose from the following options: arch, centos, debian, fedora, gentoo, mageia, mandrake, mandriva, redhat, rocks, slackware, suse, turbolinux, unitedlinux, yellowdog and darwin (for Mac OS).

Assistant for creating assistants packages There is a DevAssistant package containing set of assistants that help you create this quite complicated directory structure and package your DAP. It's called dap and you can get it form DAPI.

```
# install dap from DAPI
$ da pkg install dap

# observe available options
$ da crt dap --help

# create DAP directory structure named foo with (empty) crt and twk assistants
$ da crt dap -n foo --crt --twk

# you can also tweak your DAP directory structure a bit by adding assistants of different kind

# observe available options
$ da twk dap add -h

# add a snippet
$ da twk dap add --snippet

# once ready, you can also pack you assistant
$ da twk dap pack

# as well as check if DevAssistant thinks your package is sane
$ da pkg lint foo-0.0.1.dap
```

Uploading your DAP to DevAssistant Package Index

When you are satisfied, you can share your assistant on DAPI (DevAssistant Package Index).

On DAPI, log in with Github or Fedora account and follow Upload a DAP link in the menu.

There are two basic ways to create your own assistants. You can either learn our Yaml DSL and *write pure Yaml assistants* or you can use an approach called *"DevAssistant PingPong"*. PingPong let's you write assistants in scripting languages, while still utilizing DevAssistant functionality, so you don't have to learn the DSL (to be precise, you only need to learn a very small portion of it).

1.2.3 Command Reference

This page serves as a reference for commands of the DevAssistant *Yaml DSL*. These commands are also callable from *PingPong scripts*. Every command consists of a **command_type** and **command_input**. After it gets executed, it sets the `LAST_LRES` and `LAST_RES` variables. These are also its return values, similar to *Expressions* **logical result** and **result**.

- `LAST_LRES` is the logical result of the run - `True`/`False` if successful/unsuccessful
- `LAST_RES` is the "return value" - e.g. a computed value

In the Yaml DSL, commands are called like this:

```
- command_type: command_input
```

This reference summarizes commands included in DevAssistant itself in the following format:

`command_type` - some optional info

- Input: what should the input look like?

- RES: what is `LAST_RES` set to after this command?
- LRES: what is `LAST_LRES` set to after this command?
- Example: example usage

Note: if a command explanation says that command "*raises exception*" under some circumstances, it means that a critical error has occured and assistant execution has to be interrupted immediately. See documentation for *exceptions in run sections* for details on how this reflects on command line and in GUI. In terms of the underlying Python source code, this means that `exceptions.CommandException` has been raised.

Missing something? Commands are your entry point for extending DevAssistant. If you're missing some functionality in `run` sections, just *write a command runner* and either *include it with your assistant* or send us a pull request to get it merged in DevAssistant core.

Builtin Commands

There are three builtin commands that are inherent part of DevAssistant Yaml DSL:

- variable assignment
- condition
- loop

All of these builtin commands utilize expressions in some way - these must follow rules in *Expressions*.

Variable Assignment

Assign *result* (and possibly also *logical result*) of *Expressions* to a variable(s).

`$<var1>[, $<var2>]` - if one variable is given, *result* of expression (**command input**) is assigned. If two variables are given, the first gets assigned *logical result* and the second *result*.

- Input: an expression
- RES: *result* of the expression
- LRES: *logical result* of the expression
- Example:

```
- $foo: "bar"
- $spam:
  - spam
  - spam
  - spam
- $bar: $baz
- $success, $list~: $(ls "$foo")
```

Condition

Conditional execution.

`if <expression>, else` - conditionally execute one or the other section (`if` can stand alone, of course)

- Input: a subsection to run
- RES: RES of last command in the subsection, if this clause is invoked. If not invoked, RES remains untouched.

- LRES: LRES of last command in the subsection, if this clause is invoked. If not invoked, LRES remains untouched.
- Example:

```
- if defined $foo:
  - log_i: Foo is $foo!
- else:
  - log_i: Foo is not defined!
```

Loop

A simple for loop.

`for <var>[, <var>] [word_in,in] <expression>` - loop over result of the expression. If `word_in` is used and `<expression>` is a string, it will be split on whitespaces and iterated over; with `in`, string will be split to single characters and iterated over. For iterations over lists and mappings, `word_in` and `in` behave the same. When iterating over mapping, two control variables may be provided to get both key and its value.

- Input: a subsection to repeat in loop
- RES: RES of last command of last iteration in the subsection. If there are no interations, RES is untouched.
- LRES: LRES of last command of last iteration in the subsection. If there are no interations, RES remains untouched.
- Example:

```
- for $i word_in $(ls):
  - log_i: File: $i

- $foo:
    1: one
    2: two
- for $k, $v in $foo:
  - log_i: $k, $v
```

Ask Commands

User interaction commands, let you ask for password and various other input.

`ask_confirm`

- Input: mapping containing `prompt` (short prompt for user) and `message` (a longer description of what the user should confirm)
- RES: the confirmation (`True` or `False`)
- LRES: same as RES
- Example:

```
- $confirmed~:
  - ask_confirm:
      message: "Do you think DevAssistant is great?"
      prompt: "Please select yes."
```

`ask_input`

- Input: mapping containing `prompt` (short prompt for user)

- RES: the string that was entered by the user
- LRES: `True` if non-empty string was provided
- Example:

```
- $variable:
  - ask_input:
      prompt: "Your name"
```

`ask_password`

- Input: mapping containing `prompt` (short prompt for user)
- This command works the same way as `ask_input`, but the entered text is hidden (displayed as bullets)
- RES: the password
- LRES: `True` if non-empty password was provided
- Example:

```
- $passwd:
  - ask_password:
      prompt: "Please provide your password"
```

Command Line Commands

Run commands in subprocesses and receive their output.

`cl`, `cl_[i,r]` (these do the same, but appending `i` logs the command output on INFO level and appending `r` runs command as root; appending `p` makes DevAssistant pass subcommand error, e.g. execution continues normally even if subcommand return code is non-zero)

- Input: a string, possibly containing variables and references to files
- RES: stdout + stdin interleaved as they were returned by the executed process
- LRES: always `True`, *raises exception* on non-zero return code
- Example:

```
- cl: mkdir ${name}
- cl: cp *file ${name}/foo
- cl_i: echo "Hey!"
- cl_ir: echo "Echoing this as root"
- cl_r: mkdir /var/lib/foo
- $lres, $res:
  - cl_ip: cmd -this -will -log -in -realtime -and -save -lres -and -res -and -then -continue
```

If you need to set environment variables for multiple subsequent commands, consult *Modifying Subprocess Environment Variables*.

Note: when using `r`, it's job of DevAssistant core to figure out what to use as authentication method. Consider this an implementation detail.

A note on changing current working directory: Due to the way Python interpreter works, DevAssistant has to special-case "cd <dir>" command, since it needs to call a special Python method for changing current working directory of the running interpreter. Therefore you must always use "cd <dir>" as a single command (do not use "ls foo && cd foo"); also, using pushd/popd is not supported for now.

Modifying Subprocess Environment Variables

Globaly set/unset shell variables for subprocesses invoked by *Command Line Commands* and in *Expressions*.

`env_set`, `env_unset`

- Input: a mapping of variables to set if using `env_set`, name (string) or names (list) of variables to unset if using `env_unset`
- RES: mapping of newly set variable name(s) to their new values (for `env_set`) or unset variables to their last values (for `env_unset`)
- LRES: always `True`
- Example:

```
- env_set:
    FOO: bar
# If FOO is not in local DevAssistant context, DevAssistant does no substitution.
#  This measn that the shell still gets "echo $FOO" to execute and prints "bar".
- cl_i: echo $FOO
- env_unset: FOO
```

Note: If some variables to be unset are not defined, their names are just ignored.

Dependencies Command

Install dependencies from given **command input**.

`dependencies`

- Input: list of mappings, similar to *Dependencies section*, but without conditions and usage of sections from snippets etc.
- RES: **command input**, but with expanded variables
- LRES: always `True` if everything is ok, *raises exception* otherwise
- Example:

```
- if $foo:
  - $rpmdeps: [foo, bar]
- else:
  - $rpmdeps: []

- dependencies:
  - rpm: $rpmdeps
```

.devassistant Commands

Commands that operate with `.devassistant` file.

`dda_c` - creates a `.devassistant` file, should only be used in creator assistants

- Input: directory where the file is supposed to be created
- RES: always `True`, terminates DevAssistant if something goes wrong
- LRES: always empty string
- Example:

```
- dda_c: ${path}/to/project
```

`dda_r` - reads an existing `.devassistant` file, should be used by tweak and preparer assistants.Sets some global variables accordingly, most importantly `original_kwargs` (arguments used when the project was created) - these are also made available with `dda__` prefix (yes, that's double underscore).

- Input: directory where the file is supposed to be
- RES: always empty string
- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- dda_r: ${path}/to/project
```

`dda_w` - writes a mapping (dict in Python terms) to `.devassistant`

- Input: list with directory with `.devassistant` file as a first item and the mapping to write as the second item. Variables in the mapping will be substituted, you have to use `$$foo` (two dollars instead of one) to get them as variables in `.devassistant`.
- RES: always empty string
- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- dda_w:
  - ${path}/to/project
  - run:
    - $$foo: $name # name will get substituted from current variable
    - log_i: $$foo
```

`dda_dependencies` - installs dependencies from `.devassistant` file, should be used by preparer assistants. Utilizes both dependencies of creator assistants that created this project plus dependencies from `dependencies` section, if present (this section is evaluated in the context of current assistant, not the creator).

- Input: directory where the file is supposed to be
- RES: always empty string
- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- dda_dependencies: ${path}/to/project
```

`dda_run` - run `run` section from from `.devassistant` file, should be used by preparer assistants. This section is evaluated in the context of current assistant, not the creator.

- Input: directory where the file is supposed to be
- RES: always empty string
- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- dda_run: ${path}/to/project
```

Github Command

Manipulate Github repositories. Two factor authentication is supported out of the box.

Github command (`github`) has many "subcommands". Subcommands are part of the command input, see below.

- Input: a string with a subcommand or a two item list, where the first item is a subcommand and the second item is a mapping that explicitly specifies parameters for the subcommand.
- RES: if command succeeds, either a string with URL of manipulated repo or empty string is returned (depends on subcommand), else a string with problem description (it is already logged at WARNING level)
- LRES: `True` if the Github operation succeeds, `False` otherwise
- Example:

```
- github: create_repo

- github:
  - create_and_push
  - login: bkabrda
    reponame: devassistant

- github: push

- github: create_fork
```

Explanation of individual subcommands follows. Each subcommand takes defined arguments, whose default values are taken from global context. E.g. `create_and_push` takes an argument `login`. If it is not specified, assistant variable `github` is used.

`create_repo` Creates a repo with given `reponame` (defaults to var `name`) for a user with given login (defaults to var `github`). Optionally accepts `private` argument to create repo as private (defaults to var `github_private`).

`create_and_push` Same as `create_repo`, but it also adds a proper git remote to repository in current working dir and pushes to Github.

`push` Just does `git push -u origin master`, no arguments needed.

`create_fork` Creates a fork of repo at given `repo_url` (defaults ot var `url`) under user specified by `login` (defaults to var `github`).

Jinja2 Render Command

Render a Jinja2 template.

`jinja_render`, `jinja_render_dir` - render a single template or a directory containing more templates

- Input: a mapping containing
  - `template` - a reference to file (or a directory if using `jinja_render_dir`) in `files` section
  - `destination` - directory where to place rendered template (or rendered directory)
  - `data` - a mapping of values used to render the template itself
  - `overwrite` (optional) - overwrite the file if it exists? (defaults to `false`)
  - `output` (optional) - specify a filename of the rendered template (see below for information on how the filename is costructed if not provided), not used with `jinja_render_dir`
- RES: always `success` string

- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- jinja_render:
    template: *somefile
    destination: ${dest}/foo
    overwrite: yes
    output: filename.foo
    data:
      foo: bar
      spam: spam

- jinja_render_dir:
    template: *somedir
    destination: ${dest}/somedir
    data:
      foo: foo!
      spam: my_spam
```

The filename of the rendered template is created in this way (the first step is omitted with `jinja_render_dir`:

- if `output` is provided, use that as the filename
- else if name of the template endswith `.tpl`, strip `.tpl` and use it
- else use the template name

For template syntax reference, see Jinja2 documentation.

Logging Commands

Log commands on various levels. Logging on ERROR or CRITICAL logs the message and then terminates the execution.

`log_[d,i,w,e,c]` (the letters stand for DEBUG, INFO, WARNING, ERROR, CRITICAL)

- Input: a string, possibly containing variables and references to files
- RES: the logged message (with expanded variables and files)
- LRES: always `True`
- Example:

```
- log_i: Hello $name!
- log_e: Yay, something has gone wrong, exiting.
```

Docker Commands

Control Docker from assistants.

`docker_[build,cc,start,stop,attach,find_img,container_ip,container_name]`

- Input:
  - `attach` - list or string with names/hashes of container(s) (if string is provided, it's split on whitespaces to get names/hashes)
  - `build` - mapping with arguments same as `build` method from docker_py_api, but `path` is required and `fileobj` is ignored

  - `cc` - mapping with arguments same as `create_container` method from docker_py_api, `image` is required
  - `container_ip` - string (container hash/name)
  - `container_name` - string (container hash)
  - `find_img` - string (a start of hash of image to find)
  - `start` - mapping with arguments same as `start` method from docker_py_api, `container` is required
  - `stop` - mapping with arguments same as `stop` method from docker_py_api, `container` is required
- LRES and RES:
  - `attach` - LRES is `True` if all attached containers end with success, `False` otherwise; RES is always a string composed of outputs of all containers
  - `build` - `True` and hash of built image on success, otherwise *raises exception*
  - `cc` - `True` and hash of created container, otherwise *raises exception*
  - `container_ip` - `True` and IPv4 container address on success, otherwise *raises exception*
  - `container_name` - `True` and container name on success, otherwise *raises exception*
  - `find_img` - `True` and image hash on success if there is only one image that starts with provided input; `False` and string with space separated image hashes if there are none or more than one images
  - `start` - `True` and container hash on success, *raises exception* otherwise
  - `stop` - `True` and container hash on success, *raises exception* otherwise
- Example (build an image, create container, start it and attach to output; stop it on DevAssistant shutdown):

```
run:
# build image
- $image~:
  - docker_build:
      path: .
# create container
- $container~:
  - docker_cc:
      image: $image
# start container
- docker_start:
    container: $container
- log_i~:
  - docker_container_ip: $container
# register container to be shutdown on DevAssistant exit
- atexit:
  - docker_stop:
      container: $container
      timeout: 3
# attach to container output - this can be interrupted by Ctrl+C in terminal,
#  but currently not in GUI, see https://github.com/devassistant/devassistant/issues/284
- docker_attach: $container
```

Vagrant-Docker Commands

Control Docker using Vagrant from assistants.

`vagrant_docker`

- Input: string with vagrant command to run, must start with one of `up`, `halt`, `destroy`, `reload`
- RES: hashes/names of containers from Vagrantfile (not all of these were necessarily manipulated with, for example if you use `halt`, all container hashes are returned even if no containers were previously running)
- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- vagrant_docker: halt
- vagrant_docker: up
```

SCL Command

Run subsection in SCL environment.

`scl [args to scl command]` (note: you **must** use the scriptlet name - usually `enable` - because it might vary)

- Input: a subsection
- RES: RES of the last command in the given section
- LRES: LRES of the last command in the given section
- Example:

```
- scl enable python33 postgresql92:
  - cl_i: python --version
  - cl_i: pgsql --version
```

Note: currently, this command can't be nested, e.g. you can't run `scl enable` in another `scl enable`.

Running Commands as Another User

Run subsection as a different user (how this command runner does this is considered an implementation detail). `as <username>` (note: use `as root`, to run subsection under superuser)

- Input: a subsection
- RES: output of **the whole** subsection
- LRES: LRES of the last command in the given section
- Example:

```
- as root:
  - cl: ls /root
- as joe:
  - log_i~: $(echo "this is run as joe")
```

Note: This command invokes DevAssistant under another user and passes the whole section to it. This means some behaviour differences from e.g. `scl` command, where each command is run in current assistant. Most importantly, RES of this command is RES of all commands from given subsection.

Using Another Section

Runs a section specified by **command input** at this place.

`use` This can be used to run:

- another section of this assistant (e.g. `use: self.run_foo`)
- section of superassistant (e.g. `use: super.run`) - searches all superassistants (parent of this, parent of the parent, etc.) and runs the first found section of given name
- section from snippet (e.g. `use: snippet_name.run_foo`)
- Input: a string with section name
- RES: RES of the last command in the given section
- LRES: LRES of the last command in the given section
- Example:

```
- use: self.run_foo
- use: super.run
- use: a_snippet.run_spam
```

This way, the whole context (all variables) are passed into the section run (by value, so they don't get modified).

Another, more function-like usage is also available:

```
- use:
    sect: self.run_foo
      args:
        foo: $bar
        baz: $spam
```

Using this approach, the assistant/snippet and section name is taken from `sect` and only arguments listed in `args` are passed to the section (plus all "magic" variables, e.g. those starting and ending with double underscore).

Normalizing User Input

Replace "weird characters" (whitespace, colons, equals...) by underscores and unicode chars by their ascii counterparts.

- Input: a string or a mapping containing keys `what` and `ok_chars` (`ok_chars` is a string containing characters that should not be normalized)
- RES: a string with weird characters (e.g. brackets/braces, whitespace, etc) replaced by underscores
- LRES: True
- Example:

```
- $dir~:
  - normalize: foo!@#$%^bar_ěšč
- cl: mkdir $dir  # creates dir named foo______bar_esc
- $dir~:
  - normalize:
      what: f-o.o-@#$baz
      ok_chars: "-."
- cl: mkdir $dir  # creates dir named f-o.o-___baz
```

Setting up Project Directory

Creates a project directory (possibly with a directory containing it) and sets some global variables.

- Input: a mapping of input options, see below
- RES: path of project directory or a directory containing it, if `create_topdir` is `False`

- LRES: `True`, *raises exception* if something goes wrong
- Example:

```
- $dir: foo/bar/baz
- setup_project_dir:
    from: $dir
    create_topdir: normalized
```

Note: as a side effect, this command runner sets 3 global variables for you (their names can be altered by using arguments `contdir_var`, `topdir_var` and `topdir_normalized_var`):

- `contdir` - the dir containing project directory (e.g. `foo/bar` in the example above)
- `topdir` - the project directory (e.g. `baz` in the example above)
- `topdir_normalized` - normalized name (by *Normalizing User Input*) of the project directory

Arguments:

- `from` (required) - a string or a variable containing string with directory name (possibly a path)
- `create_topdir` - one of `True` (default), `False`, `normalized` - if `False`, only creates the directory containing the project, not the project directory itself (e.g. it would create only `foo/bar` in example above, but not the `baz` directory); if `True`, it also creates the project directory itself; if `normalized`, it creates the project directory itself, but runs it's name through *Normalizing User Input* first
- `normalize_ok_chars` - string containing characters that should not be normalized, assuming that `create_topdir: normalized` is used
- `contdir_var`, `topdir_var`, `topdir_normalized_var` - names to which the global variables should be assigned to - *note: you have to use variable names without dollar sign here*
- `accept_path` - either `True` (default) or `False` - if `False`, this will terminate DevAssistant if a path is provided
- `on_existing` - one of `fail` (default), `pass` - if `fail`, this will terminate DevAssistant if directory specified by `from` already exists; if `pass`, nothing will happen; note, that this is always considered `pass`, if `create_topdir` is `False` (in which case the assistant is in full control and responsible for checking everything itself)

Running Commands After Assistant Exits

Register commands to be run when assistant exits (this is not necessarily DevAssistant exit).

- Input: section (list of commands to run)
- RES: the passed list of commands (raw, unformatted)
- LRES: True
- Example:

```
- $server: $(get server pid)
- atexit:
  - cl: kill $server
  - log_i: Server gets killed even if the assistant failed at some point.'
```

Sections registered by `atexit` are run at the very end of assistant execution even after the `post_run` section. There are some differencies compared to `post_run`:

- `atexit` command creates a "closure", meaning the values of variables in time of the actual section invocation are the same as they were at the time the `atexit` command was used (meaning that even if you change variable values during the `run` section after running `atexit`, the values are preserved).
- You can use multiple `atexit` command calls to register multiple sections. These are run in the order in which they were registered.
- Even if some of the sections registered with `atexit` fail, the others are still invoked.

DevAssistant PingPong

Run *DevAssistant PingPong scripts*.

- Input: a string to line on commandlie
- RES: Result computed by the PingPong script
- LRES: Logical result computed by the PingPong script
- Example:

```
- pingpong: python3 *file_from_files_section
```

Loading Custom Command Runners

Load DevAssistant *command runner(s)* from a file.

- Input: string or mapping, see below
- RES: List of classnames of loaded command runners
- LRES: True if at least one command runner was loaded, False otherwise
- Example:

```
files:
  my_cr: &my_cr
    source: cr.py

run:
- load_cmd: *my_cr
# assuming that there is a command runner that runs "mycommand" in the file,
#  we can do this as of now until the end of this assistant
#  this is equivalent of
#  - load_cmd:
#      from_file: *my_cr
- mycommand: foo

# load command runner from file provided in hierarchy of a different assistant
# - make it prefixed to make sure it doesn't conflict with any core command runners
# - load only BlahCommandRunner even if the file includes more runners
- load_cmd:
    from_file: crt/someotherassistant/crs.py
    prefix: foo
    only: BlahCommandRunner
- foo.blah: input  # runs ok
- blah: input  # will fail, the command runner was registered with "foo" prefix
```

Note: since command runners loaded by `load_cmd` have higher priority than DevAssistant builtin command runners, you can use this to *override* the builtins. E.g. you can have a command runner that overrides `log_i`. If someone

wants to use this command runner of yours but also keep the original one, he can provide a `prefix`, so that your logging command is only available as `some_prefix.log_i`.

1.2.4 Project Metainfo: the .devassistant File

Note: .devassistant file changed some of its contents and semantics in version 0.9.0.

Project created by DevAssistant usually get a `.devassistant` file, see *.devassistant Commands* for information on creating and manipulating it by assistants. This file contains information about a project, such as project type or paramaters used when this project was created. It can look like this:

```
devassistant_version: 0.9.0
original_kwargs:
  name: foo
  github: bkabrda
project_type: [python, django]
dependencies:
- rpm: [python-django]
```

When .devassistant is used

Generally, there are two use cases for `.devassistant`:

- Tweak assistants read the `.devassistant` file to get project type (which is specified by `project_type` entry) and decide what to do with this type of project (by choosing a proper `run` section to execute and proper `dependencies` section, see *Tweak Assistants*).
- When you use the `custom` preparer with URL to this project (`da prepare custom -u <url>`), DevAssistant will checkout the project, read the data from `.devassistant` and do few things:
  - It will install any dependendencies that it finds in `.devassistant`. These dependencies look like normal *dependencies section* in assistant, e.g.:

    ```
    dependencies:
    - rpm: [python-spam]
    ```

  - It will also run a `run` section from `.devassistant`, if it is there. Again, this is a normal *run section*:

    ```
    run:
    - log_i: Hey, I'm running from .devassistant after checkout!
    ```

  Generally, when using `custom` assistant, you have to be **extra careful**, since someone could put `rm -rf ~` or similar evil command in the `run` section. So use it **only with projects whose upstream you trust**.

1.2.5 Project Types

This is a list of official project types that projects should use in their `.devassistant` file and Creator assistants should state. If you choose one of the official project types, there is a good chance that Tweak and Preparer assistants written by others will work well with projects created by your Creator.

The project type is given as a list of strings - these describe the project from the most general type to the most specific. E.g:

```
project_type: [python, django]
```

If you don't use `project_type` in your Creator assistant, it will be automatically generated: If your assistant is `crt/footest/foobar.yaml`, project type in `.devassistant` will be `[footest, foobar]`. This means that Tweak and Preparer assistants written by others may not work well with your project, but otherwise it does no harm.

Current List of Types

Current project types list follows. If you want anything added in here, open a bug for us at https://github.com/devassistant/devassistant/issues. **Note: the list is currently not very thorough and it is meant to grow as we get requested by assistant developers.**

- c
- cpp
- java
- nodejs
  - express
- perl
  - dancer
- php
- python
  - django
  - flask
  - gtk3
  - lib
- ruby
  - rails

1.2.6 Contributing to DevAssistant

We are very happy that you want to contribute to DevAssistant, and we want to make this as easy as possible for you - that's what DevAssistant is all about anyway. To save both you and ourselves a lot of time and energy, here we list some rules we would like you to follow to make the pull request process as quick and painless as possible.

Have a look at our code first

Every programmer has a different programming style, a different way of thinking, and that's good. However, if several people contribute to the same project, and each one of them keeps to their style while ignoring the others, it becomes very hard to read the code afterwards. Please, before you start coding your solution, have a look at similar parts of DevAssistant's code to see how we approached it, and try to follow that if possible. You will make future maintenance much easier for everyone, and we will be able to review your pull requests faster as well.

Use PEP8

We follow PEP8, and we ask you to do that as well. It makes the code much more readable and maintainable. Our only exception is that lines can be as long as 99 characters.

Write tests

Good code has tests. The code you wrote works now, but once someone changes something, it may all break apart. There are a few general good practices to go by if you're writing code:

- If you write some new feature, please write tests that make sure it works when everything is okay, and that it fails the expected way when it isn't.
- If you fix something, please create tests that ensure that the code really works the new way, and that it doesn't work the way it used to work before.

If you go by these rules, there is very little chance that your code breaks some other part of DevAssistant, and at the same time, you make your part of code less likely to break in the future.

For testing, we use pytest.

When testing, use mocking (namely flexmock)

Often when you need to test some object's behaviour, you need to "pretend" that something works somehow, for example that the network is up or that a specific file exists. That is okay, but it is not okay to actually connect to the internet for testing, or create or delete specific files in the file system. This could break something, or might not work on our test server.

Of course, sometimes you may need to create a nameless temporary file with `tempfile.mkstemp()`, which is something we do often, and it is a perfectly acceptable practice. However, you should not touch for example the `~/.devassistant/config` file, which actually belongs to the user, and by writing it, you could delete or damage the user's config.

To overcome these problems, we are using flexmock, which is a library that allows you to modify the behaviour of the environment so that you don't have to rely on the values on the user's machine. By calling flexmock on an object, you can either change some of its methods or attributes, or you can completely replace it with a flexmock object whose behaviour you fully control.

An example:

```
import os
from flexmock import flexmock

def test_something(self):
    flexmock(os.path).should_receive('isfile').with_args('/foo/bar/baz').and_return(True)
    do_something_assuming_foobarbaz_is_a_valid_file()
```

What you did here is modify the behaviour of the method `os.path.isfile()` so that it returns `True` when called with the argument `/foo/bar/baz`. This works only within the current code block, so you can mock something in one test, and then just forget about it. The next test will have clean environment again.

Here is flexmock documentation.

Just a note here: Mocking doesn't work well in setup and teardown methods, because they are different code blocks.

Parameterize tests

It makes perfect sense to feed multiple values to a method to see how it works in different situations. Very often it's done like this:

```
def test_something(self):
    for value, number in [('foo', 1), ('bar', 2), ('baz', 3)]:
        do_something(value, number)
```

That's not exactly how we want to do it. For one, if it fails, you can't quickly see what the values were when the test failed, so you have to use a debugger or put some print statements in the code. Another thing is that it's harder to read and more prone to error. The preferred way of achieving the same functionality is this:

```
@pytest.mark.parametrize(('value', 'number'), [
    ('foo', 1),
    ('bar', 2),
    ('baz', 3),
])
def test_something(self, value, number):
    do_something(value, number)
```

The second example is much better especially if you're doing more than just calling one method - for example mocking, running a setup/teardown method etc. Pytest also automatically outputs the test parameters if a test fails, so debugging is much easier. We strongly encourage you to use the second example, and might not accept your pull request if you don't, unless you present a good reason why.

Use six for Python 2 + 3 compatibility

DevAssistant works with both major versions of Python currently in production, and we want to keep only one codebase, therefore we need an interoperability library, namely `six`. This library is much more powerful and easy to use than, say, importing *__future__*, so please, use `six` and nothing else.

In a majority of cases, we use `six` for these things:

- importing libraries that were moved or renamed
- testing if a variable contains a string/unicode/bytes
- testing what version of python DevAssistant is running on.

To import a library that was renamed in Python 3, you use the `six.moves.builtins` module:

```
from six.moves.builtins import urllib
```

This imports a module mimicking Python 3's *urllib* module, so both in Python 2 and Python 3, you then call:

```
urllib.request.urlretrieve(url)
```

The variable containing the information if the code is running under Python 3 is found here:

```
import six
six.PY3
```

There is also the `six.PY2` constant, but that was added to `six` quite recently, so for better backwards compatibility, we kindly ask you to use `not six.PY3` instead.

Use pyflakes to sanitize your code

Pyflakes (as well as pylint), are two great tools for improving the quality of your code. We especially urge you to use pyflakes to find unused imports, undeclared variables and other errors detectable without actually running the code.

Always talk to us when:

Your contribution changes dependencies

We try to keep DevAssistant's dependency chain as small as possible, so if your code adds a dependency, it is a big deal for us. For this reason, we urge you to talk to us first (here's how). If we decide that the new dependency is necessary, we'll gladly give you a green light and accept your contribution. If we think that your idea can do without adding the new package, we'll do our best to help you modify your idea.

However, if you do not talk to us and implement your feature right away, there is a risk that we will reject your contribution and you will have to throw your existing code away and start from scratch.

You want to implement a large feature

We welcome large contributions, and are very happy that you take the interest and time to make them. However, we have certain plans where DevAssistant should go, or what it should look like, and there's quite a good chance that if you don't discuss your idea with us, you might write something quite different, which we won't be willing to accept.

To avoid this kind of situations, always consult your intentions with us before you start coding - we're more than open to new ideas, but we want to know about them first.

You want to include your contribution in an upcoming release

We do have a release plan, but this doesn't mean we couldn't occasionally wait a few days for your feature to be included. If you tell us about your contribution, and we decide that we want it in, we'll hold a release for you to finish and submit your code. Of course, the sooner you tell us, the better the outcome will be.

1.2.7 Talk to Us!

If you want to see where DevAssistant development is going and you want to influence it and send your suggestions and comments, you should join our ML: https://lists.fedoraproject.org/mailman/listinfo/devassistant. We also have IRC channel #devassistant on Freenode and you can join our Google+ community.

1.2.8 Overall Design

DevAssistant consists of several parts:

Core Core of DevAssistant is written in Python. It is responsible for interpreting Yaml Assistants and it provides an API that can be used by any consumer for the interpretation.

CL Interface CL interface allows users to interact with DevAssistant on commandline; it consumes the Core API.

GUI (work in progress) GUI allows users to interact with Developer Assistant from GTK based GUI; it consumes the Core API.

Assistants Assistants are Yaml files with special syntax and semantics (defined in *Yaml DSL Reference*). They are indepent of the Core, therefore any software distribution can carry its own assistants and drop them into the directory from where DevAssistant loads them - they will be loaded on next invocation. Note, that there is also a possibility to write assistants in Python, but this is no longer supported and will be removed in near future.

1.2.9 Assistants

Internally, each assistant is represented by instance of `devassistant.yaml_assistant.YamlAssistant`. Instances are constructed by DevAssistant in runtime from parsed yaml files. Each assistant can have zero or more subassistants. This effectively forms a tree-like structure. For example:

```
       MainAssistant
       /           \
      /             \
    Python          Ruby
    /   \           / \
   /     \         /   \
Django  Flask    Rails Sinatra
```

This structure is defined by filesystem hierarchy as explained in *Assistants Loading Mechanism*

Each assistant can optionally define arguments that it accepts (either on commandline, or from GUI). For example, you can run the leftmost path with:

```
$ da create python [python assistant arguments] django [django assistant arguments]
```

If an assistant has any subassistants, one of them **must** be used. E.g. in the example above, you can't use just Python assistant, you have to choose between Django and Flask. If Django would get a subassistant, it wouldn't be usable on its own any more, etc.

Assistant Roles

The `create` in the above example means, that we're running an assistant that creates a project.

There are four assistant roles:

creator (`create` or `crt` on command line) creates new projects

tweak (`tweak` or `twk` on command line) works with existing projects

preparer (`prepare` or `prep` on command line) prepares environment for development of upstream projects

extras (`extras` or `extra` on command line) performs arbitrary tasks not related to a specific project

The main purpose of having roles is separating different types of tasks. It would be confusing to have e.g. `python django` assistant (that creates new project) side-by-side with `eclipse` assistant (that registers existing project into Eclipse).

1.2.10 Writing Assistants: Yaml or Scripting Languages

There are two ways to write assistants. You can either use our *Yaml based DSL* or write assistants in popular scripting languages (for list of supported languages see *Supported Languages*). This method is referred to as *DevAssistant PingPong*.

1.2.11 Contributing

If you want to contribute (bug reporting, new assistants, patches for core, improving documentation, ...), please use our Github repo:

- code: https://github.com/devassistant/devassistant
- issue tracker: https://github.com/devassistant/devassistant/issues

If you have DevAssistant installed (version 0.8.0 or newer), there is a fair chance that you have `devassistant` preparer. Just run `da prepare devassistant` and it will checkout our sources and do all the boring stuff that you'd have to do without DevAssistant.

If you don't have DevAssistant installed, you can checkout the sources like this (just copy&paste this to get the job done):

```
git clone https://github.com/devassistant/devassistant
```

You can find list of core Python dependencies in file `requirements.txt`. If you want to write and run tests (you should!), install dependencies from `requirements-devel.txt`:

```
pip install --user -r requirements-devel.txt
```

If you develop on Python 2, you'll also need to install extra dependencies:

```
pip install --user -r requirements-py2.txt
```

Regardless of Python version, you'll need `polkit` for requesting root privileges for dependency installation etc. If you want to play around with GUI, you have to install `pygobject`, too. To run guitest, you also need to install behave from PyPI and dogtail (not on PyPI, get it from Fedora Hosted or from your favorite package manager). (See how hard this is compared to `da prepare devassistant`?)

CHAPTER 2

Overview

This is documentation for version 0.10.0.

DevAssistant - start developing with ease

DevAssistant (http://devassistant.org) can help you with creating and setting up basic projects in various languages, installing dependencies, setting up environment etc.

It is based on idea of per-{language/framework/...} "assistants" (plugins) with hierarchical structure.

Note: prior to version 0.10.0, DevAssistant has been shipped with a default set of assistants that only worked on Fedora. We decided to drop this default set and create DAPI, DevAssistant Package Index, https://dapi.devassistant.org/ *- an upstream PyPI/Rubygems-like repository of packaged assistants. DAPI's main aim is to create a community around DevAssistant and provide various assistants with good support for various platforms - a task that DevAssistant core team alone is not able to achieve for a large set of assistants.*

This all means that if you get DevAssistant from upstream repo or from PyPI, you will have no assistants installed by default. To get assistants, search DAPI through web browser or run `da pkg search <term>` *and* `da pkg install <assistant package>`. *This will install one or more DAPs (DevAssistant Packages) with the desired assistants.*

If you want to create your own assistants and upload them to DAPI, see http://docs.devassistant.org/en/latest/developer_documentation/create_assistant.html and http://docs.devassistant.org/en/latest/developer_documentation/create_assistant/packaging_and_distributing.html.

There are four main modes of DevAssistant execution. Explanations are provided to better illustrate what each mode is supposed to do:

`create` Create new projects - scaffold source code, install dependencies, initialize SCM repos ...

`tweak` Work with existing projects - add source files, import to IDEs, push to GitHub, ...

`prepare` Prepare environment for working with existing upstream projects - install dependencies, set up services, ...

`extras` Tasks not related to a specific project, e.g. enabling services, setting up IDEs, ...

These are some examples of what you can do:

```
# search for assistants that have "Django" in their description
$ da pkg search django
python - Python assistants (library, Django, Flask, GTK3)

# install the found "python" DAP, assuming it supports your OS/distro
$ da pkg install python

# find out if the installed package has documentation
$ da doc python
```

```
INFO: DAP "python" has these docs:
...
INFO: usage.txt
...
# show help
$ da doc python usage.txt

# if the documentation doesn't say it specifically, find out if there is a "create"
#  assistant in the installed "python" DAP
$ da create -h
...
{..., python, ...}
...

# there is, so let's find out if it has any subassistants
$ da create python -h
...
{..., django, ...}
...

# we found out that there is "django" subassistant, let's find out how to use it
$ da create python django -h
<help text with commandline options>

# help text tells us that django assistant doesn't have subassistants and is runnable, let's do it
$ da create python django -n ~/myproject # sets up Django project named "myproject" inside your home

# using the same approach with "pkg search", "pkg install" and "da tweak -h",
#  we find, install and read help for "tweak" assistant that imports projects to eclipse
$ da tweak eclipse -p ~/myproject # run in project dir or use -p to specify path

# using the same approach, we find, install and read help for assistant
#  that tries to prepare environment for a custom upstream project, possibly utilizing
#  its ".devassistant" file
$ da prepare custom -u scm_url -p directory_to_save_to

# sometimes, DevAssistant can really do a very special thing for you ...
$ da extras make-coffee
```

Should you have some questions, feel free to ask us at Freenode channel #devassistant or on our mailing list (https://lists.fedoraproject.org/mailman/listinfo/devassistant). You can also join our G+ community (https://plus.google.com/u/0/communities/112692240128429771916) or follow us on Twitter (https://twitter.com/dev_assistant).

DevAssistant works on Python 2.6, 2.7 and >= 3.3.

This whole project is licensed under GPLv2+.